architecte logiciel

Antoine Lonjon • Jean-Jacques Thomasson
Ouvrage dirigé par Libero Maesano

Modélisation XML

EYROLLES

architecte logiciel

Modélisation XML

architecte logiciel

Antoine **Lonjon** • Jean-Jacques **Thomasson**
Ouvrage dirigé par Libero **Maesano**

Modélisation XML

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Le code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.
© Groupe Eyrolles, 2006, ISBN : 2-212-11521-0

Préface

En 2000 a circulé sur Internet une histoire qui s'est révélée par la suite une imbrication inextricable d'informations vraies et de légendes. Elle peut être résumée comme suit.

L'écartement standard entre les rails aux États-Unis est de 4 pieds et 8,5 pouces, ce qui est une mesure particulièrement biscornue. Pourquoi cette mesure ? Parce que les premières lignes de chemin de fer ont été bâties principalement par des expatriés anglais et sûrement avec de la technologie et des outils anglais, les Anglais étant à l'époque à la pointe de la technologie. Pourquoi les Anglais ont-ils exporté cette mesure ridicule ? Parce que les premiers chemins de fer ont été construits par les hommes et avec les outils des fabricants de tramways sur route, et donc la distance entre les roues de ces tramways est devenue l'entre-rails. Et pourquoi cette distance entre les roues des tramways ? Parce que chaque fois qu'une autre distance avait été mise en œuvre dans le passé, le tramway, ou tout autre véhicule, s'était « cassé la figure » à cause de profonds sillons, sortes de rails, creusés à cette distance et présents sur toutes les routes d'Angleterre et d'Europe. Pourquoi des ornières avec cette distance entre elles ? Parce que ces routes avaient été d'abord bâties pour faciliter les déplacements des légions de l'empire romain, et les chariots de guerre romains, leurs principaux usagers, exhibaient une telle distance entre les roues.

À partir d'ici, l'histoire se perd un peu, et on évoque les dimensions des postérieurs des chevaux pour expliquer la distance entre les roues des chariots de guerre. Quoi qu'il en soit, c'est à partir d'une spécification des ingénieurs de l'armement de Rome que, au troisième millénaire et aux États-Unis, la voie standard des chemins de fer est spécifiée. L'histoire ensuite prend un tournant particulièrement technologique, avec les dimensions des *solid rocket boosters* de la navette spatiale, qui seraient définies ainsi

parce que les SRB doivent être transportés par chemin de fer, à travers des tunnels dont la largeur est à peine supérieure à l'écartement des rails, de l'usine Thiokol (Utah) au site de lancement… La conclusion péremptoire est que les dimensions d'un des systèmes de transport les plus modernes et avancés du point de vue technologique sont déterminées par la largeur des postérieurs des chevaux des légions romaines !

Cette histoire est une légende (http://www.snopes.com/history/american/gauge.htm, http://truthorfiction.com/rumors/r/railwidth.htm), mais sa morale est édifiante : « *Specifications and bureaucracies live forever* ». En d'autres termes, nous manifestons la tendance forte, par conformisme (diraient les pessimistes) ou par sens pratique (diraient les optimistes), à garder en vie certains éléments des architectures que nous construisons dans tous les domaines au-delà de toute attente. Ces éléments ne sont pas matériels (on passe des chariots de guerre à la navette spatiale, en passant par les tramways, les chemins de fer…), mais bien immatériels : c'est des *mesures*, des *formats*, des *protocoles*, des *spécifications*, des choses que les anglo-saxons appellent efficacement « software », ce qui ne veut pas dire simplement « code de programmation », mais se définit par opposition à l'« hardware », qui ne veut pas dire simplement « ordinateur ». C'est à première vue étonnant de se rendre compte que beaucoup de choses « soft » sont si dures à mourir, beaucoup plus que le « hard » qui est en dessous. Si on y réfléchit, il est pourtant clair que le « hard » est matériel et donc certainement périssable, alors que le « soft » est immatériel et donc a priori immortel.

Lorsque la base de données stratégique était le point de rencontre de toutes les applications, son schéma était l'objet de toutes les attentions, l'enjeu fondamental de la maîtrise du système d'information. À l'époque des systèmes répartis et interconnectés, du Web et des services Web, les schémas XML sont les « spécifications » du format de l'information. La légende nous apprend que ces spécifications dureront beaucoup plus longtemps que les machines, les réseaux, les bases, les systèmes et les programmes qui interprètent, transportent, stockent, manipulent et présentent l'information. En plus, elles détermineront de façon essentielle la capacité à interopérer des dits systèmes et programmes.

Le livre dense et intéressant de Jean-Jacques et Antoine nous fait bénéficier de leur grande expérience et compétence, pour nous expliquer comment bâtir au mieux nos « spécifications » de l'information. Je dis au mieux, car il est clair, et le livre insiste là-dessus (c'est même son leit-motiv), que la spécification parfaite est précisément et modestement le meilleur compromis entre exigences contradictoires. C'est pour cela que le métier d'architecte de l'information n'est pas facile, mais peut s'appuyer efficacement sur des méthodes et techniques de modélisation qui lui simplifient partiellement la vie.

On pourrait dire : d'accord, mes schémas ne sont pas bons, je peux les changer facilement, finalement tout cela ce n'est que du soft ! La morale de l'histoire est que les spécifications qui s'affirment, qu'elles soient bonnes ou non, sont destinées à rester longtemps, car le nombre de gens qui se les approprient et bâtissent par-dessus devient vite non maîtrisable et après c'est trop tard. Cette vitesse devient foudroyante lorsque les systèmes en question ne manipulent pas la matière ou l'énergie, mais l'information et, en plus, sont immergés dans l'économie de réseau, à l'intérieur comme à l'extérieur de l'entreprise. Les spécifications, soumises à l'« effet réseau » (plus on les utilise, plus elles deviennent intéressantes, indépendamment de leur qualité, et on les utilise encore plus car elles sont intéressantes) deviennent inévitablement des verrouillages (*lock-in*). La qualité des spécifications n'allonge pas leur vie, mais diminue l'inconfort et la fatigue des usagers et des développeurs, ce qui devrait être considéré par tout professionnel comme un objectif de nature déontologique.

Que le lecteur ne se trompe pas : la qualité des modèles et des schémas XML est beaucoup plus importante, par son incidence sur les systèmes d'information, que toute architecture matérielle, logicielle, de processus, car ces dernières peuvent être beaucoup plus facilement modifiées, adaptées, remplacées. Donc, si l'on doit choisir, c'est là qu'il faut dédier la plus grande attention et l'ouvrage de Jean-Jacques et Antoine est le livre de chevet de celui qui en a la lourde tâche.

Table des matières

Avant-propos XIX

Introduction générale 1

Pourquoi des modèles ? 2
UML et XML : un duo gagnant 4
UML et les différentes vues d'un système d'information 5
XML au cœur des systèmes d'information 7
Apports de XML à la modélisation 7
L'universalité de XML 8
L'interopérabilité des documents XML 9
L'indépendance entre modèles et données 9
La forme des modèles XML 10
Les apports du travail de normalisation 11
Un facteur d'adaptation des applications 13
L'adressage 14
Le stockage 14
L'archivage 16
L'expression 17
La pérennité et la flexibilité 19
En résumé 21

PREMIÈRE PARTIE
Étapes de la démarche de modélisation 23

CHAPITRE 1
Étape 1 - La préparation du projet 25

Bref aperçu de la méthode 26
Objectifs 26
Principes généraux 26

Limites 27
Contexte idéal d'utilisation de la méthode 28
Présentation détaillée des quatre principes de base de la méthode 28
Principe n°1. Un système d'information repose toujours sur un même modèle de base. 28
Principe n°2. Un système d'information est un ensemble de fonctions et services. 29
Principe n°3. Un système d'information est obligatoirement flexible. 31
Principe n°4. La représentation du système d'information doit être synthétique et simple. 33
Mise en œuvre de la méthode 34
Étape n°1. Établir un plan de classification des fonctions et services du système 35
Étape n°2. Identifier et classer les fonctions du système 37
Étape n°3. Identifier et classer les services du système 39
Étape n°4. Comparer les scénarios d'implémentation 40
En résumé… 45

CHAPITRE 2
Étape 2 - Réaliser le modèle conceptuel 47
Différence entre modèle hiérarchique et modèle d'association 48
Modèle d'objet UML et document XML 50
Modèle d'objets UML 51
Modèle de base des documents XML : Infoset 51
Notion de classe 55
Modèles et métamodèles 56
Modèles de données conceptuels, logiques et physiques 58
Le cas PiloteWeb 59
Présentation du cas 59
Modèle conceptuel de données de PiloteWeb 60
Découverte des principaux concepts 60
Découverte des principales relations 61
Recherche et formalisation des concepts et relations intermédiaires 64
Synthèse 67
En résumé… 68

CHAPITRE 3
Étape 3 - Réaliser les modèles logiques 69
Découpage du modèle conceptuel en modèles logiques 70

Identifier les objets majeurs et leur périmètre 71
Analyse des associations 71
Analyse des relations de généralisation/spécialisation 75
Modèles et espace de noms 84
Classe, type XML et élément XML 88
Gestion des associations 91
Les associations de composition 93
Les associations d'agrégation 96
Introduction 96
Règles par défaut 96
Prise en compte des modules de données 99
Les associations simples 101
Gestion des attributs 105
Le schéma logique de l'application PiloteWeb 106
Module de données « sites » 106
Module de données « pilotes » 108
Module de données « aerodromes » 109
Module de données « partenaires » 111
Module de données « restaurants » 112
Module de données « loueurs » 114
Module de données « aeroclubs » 115
Le schéma relationnel de l'application PiloteWeb 116
En résumé… 117

CHAPITRE 4
Étape 4 - Spécifier les modèles de stockage **119**
Réaliser un modèle physique, démarche générale 120
Définir la manière dont les données seront physiquement stockées 121
Choisir une forme de stockage 121
Un seul schéma et un seul document 121
Un seul schéma et plusieurs documents 122
Plusieurs schémas et un seul document par schéma 123
Plusieurs schémas et plusieurs documents par schéma 123
Choisir une solution de stockage 124
Stockage dans le système de fichiers 124
Stockage dans une base de données relationnelle 125
Stockage dans une base de données XML 139
Construire la stratégie d'adressage 142
Cas d'école : PiloteWeb 143

Découpage initial du modèle logique 143
Stockage dans un système de fichiers 147
Stockage en relationnel 150
Vues représentant l'arborescence des fichiers 151
Vues relatives aux schémas 157
La récupération des fichiers XML 162
La mise à jour d'un fichier XML 164
Stockage dans une base de données XML 165
En résumé… 169

CHAPITRE 5
Étape 5 - Finaliser la sémantique du balisage........................ 171
Principes de base 172
Type et sémantique 173
Exploiter le type et la sémantique 174
Manières d'exprimer la sémantique 177
Adapter les structures au contexte 181
Outrepasser le principe des objets métier Java 182
Utilisation des attributs pour passer la sémantique 185
En résumé… 188

CHAPITRE 6
Étape 6 - Produire les variantes des schémas XML................ 189
Pourquoi des variantes ? 189
Raisons liées à la nature de XML 189
Raisons liées à la nature de l'information 190
Un cas concret 191
Comment identifier les variantes ? 191
Schéma sans espace de noms cible 192
Schéma avec un espace de noms cible 192
Le schéma variant est une copie du schéma initial 193
Le schéma variant est une redéfinition du schéma initial 194
La variante est obtenue par inclusion du schéma initial et dérivation de ses types 195
Conséquence de la création d'une variante d'un sous-schéma 198
Conséquence des variantes sur les liens 199
En résumé… 201

CHAPITRE 7
Étape 7 - Organiser les différentes couches de programmation. 203
Présentation générale 204
D'un modèle de stockage à un modèle de présentation 209
Organisation du modèle de stockage 209
Schémas sites et pilotes 211
Organisation des modèles d'affichage 214
En synthèse 216
Les différentes couches de programmation 217
Mise en correspondance fonctionnelle avec XQuery 217
Programmes du niveau entreprise (Java) 222
Programmes du niveau métier (JSP) 225
Programmation de la présentation (XSLT) 227
Programmation de l'affichage (HTML et CSS) 230
En résumé… 231

DEUXIÈME PARTIE
Modèles de référence 233

CHAPITRE 8
Modèles modulaires 235
Représentation d'un modèle modulaire 236
Cas des DTD 239
Choix des éléments racines 239
Composition d'éléments 241
Cas des schémas XML 246
Choix des éléments racines 246
Composition d'éléments 248
En résumé… 252

CHAPITRE 9
Éléments purement structurels 253
Genèse de la théorie des éléments purement structurels 254
Peut-on assimiler une balise XML à un fichier ? 255
Cas des éléments simples 256
Cas des éléments complexes 257
Cas des éléments mixtes 258
Cas des éléments vides 260

Où doit-on mettre les attributs ? 261
Que faire des métadonnées ? 263
Les éléments XML ne sont pas tous structurellement égaux 265
Découvrir les éléments purement structurels 266
Règles d'identification 266
Exemples d'application 267
Méthode pratique d'application des règles 271
Étape 1. Préparation de l'environnement 272
Étape 2. Supprimer du schéma les commentaires et toutes les définitions d'attributs ou de notations 273
Étape 3. Création de trois éléments vides, SDE, GDE et PSE 273
Étape 4. Remplacement de tous les éléments ayant un contenu mixte par l'élément SDE 274
Étape 5. Remplacement de tous les éléments vides par l'élément SDE . . . 276
Étape 6. Remplacement de tous les éléments simples par l'élément SDE . 276
Étape 7. Transformation en GDE des éléments dont le modèle de contenu est une série d'éléments SDE 277
Étape 8. Transformation en PSE des éléments dont le modèle de contenu est une série d'éléments GDE 278
Étape 9. Remplacement de tous les éléments qui n'ont qu'un seul enfant possible 279
Étape 10. Remplacement en GDE de tous les connecteurs xs:choice qui contiennent des éléments SDE mêlés à d'autres 280
Résultat final 281
En résumé… 285

CHAPITRE 10
Concevoir des modules d'information 287
Définir un module d'information 288
Créer des modules à la bonne taille 289
Les racines et le feuillage : les deux parties de l'arbre 295
Vérifier la nature des éléments racines des modules 296
Créer les articulations de la structure 296
Règles de conception applicables aux modules 300
Règle de non-emboîtement 301
Règle d'applicabilité 303
Règle de balisage 306
Règle de limite inférieure de modularisation 308
Règle d'identification des éléments 308

Les structures d'assemblage 311
Exemple concret 315
Identification des modules 316
Classification des modules 317
Structure d'assemblage 318
En résumé 321

CHAPITRE 11
Modèles pour la gestion des métadonnées **323**
Métadonnées définies par le schéma XML 324
Métadonnées relatives à l'entité document 329
Métadonnées spécifiques d'une application métier : cas de la S1000D 329
Métadonnées spécifiques d'un média particulier : cas de xHTML 333
Métadonnées définies par le Dublin Core 335
Métadonnées définies par RDF 341
Éléments complémentaires de RDF 346
Métadonnées relatives au contenu du document 352
Métadonnées transmises par un attribut 352
Métadonnées transmises par une structure d'éléments 353
En résumé 356

CHAPITRE 12
Modèles pour la gestion des liens **359**
Les différents types de liens 360
Liens logiques et physiques 361
Liens physiques 361
Liens logiques 362
Exemple de mise en œuvre d'un lien logique 363
Cas extrême 367
Liens simples de type ID/IDREF 367
L'approche du W3C : XLink 368
Introduction 368
Revue des éléments de XLink servant à spécifier des liens 370
Modèle conceptuel de XLink 370
Élément Xlink de type lien simple 371
Élément Xlink de type étendu (ou extended) 373
Élément Xlink de type ressource 374
Élément Xlink de type localiseur (ou locator) 374
Élément Xlink de type arc 375

Élément Xlink de type titre (ou title) 376
Exemple de mise en œuvre de XLink 377
Lien de type simple 377
Lien de type complexe 378
L'approche de l'ISO : le modèle Topics Map 380
Brève description 382
Relation entre topiques et ressources physiques 383
Les associations de topiques 385
Les occurrences de topiques 389
Carte de topiques déduites du balisage 390
Questions relatives à la gestion des liens 391
Liens et gestion des révisions 391
Le risque : la paralysie du système 392
La solution 392
Liens et gestion de configuration 393
Mécanismes permettant de contrôler les liens partir d'un schéma 396
En résumé 400

CHAPITRE 13
Modèles pour la gestion des révisions et des versions.......... 401
Gestion des révisions à l'intérieur d'un document XML 402
Introduction 402
Les possibilités de balisage des révisions 402
Utilisation d'attributs 402
Utilisation de balises fixes 403
Utilisation de balises flottantes 404
Les difficultés du balisage des révisions 406
Balisage contraint par le balisage principal 406
Révision des attributs 408
Les liens 408
Difficultés pour les auteurs 408
Un exemple de balisage 409
Structure racine 409
Structure de l'élément mfmatr 410
Structure de l'élément chapter 411
Structure de l'élément increv 411
Structure de l'élément tr 412
Le bornage précis des révisions 413
Exemple basé sur l'utilisation des URN 416

Introduction aux URN 417
Liens réalisés au moyen des URN 419
Service Web pour résoudre les URN 421
Exemple concret 422
Autres méthodes de gestion des révisions 425
L'approche base de données XML 425
L'approche par calcul différentiel XUpdate 426
L'approche base de données relationnelle 427
En résumé 430

ANNEXE A
Représentation UML avancée pour XML Schema **431**
Attributs et types listes de XML Schema 432
Attributs et valeur par défaut 435
La définition d'annotations 435
Groupe d'attributs 435
La mixité des modèles 438
Contrainte d'exclusion et groupe de choix 439
Contraintes de simultanéité et groupes simples 440
La question de l'ordre des éléments 443

ANNEXE B
Ressources **445**
Textes normatifs 445
Bibliographie 453

ANNEXE C
Sigles et acronymes **455**

ANNEXE D
Infoset **457**
Unité d'information de type document 457
Unité d'information de type élément 458
Unité d'information de type attribut 459
Unité d'information de type instruction de traitement 459
Unité d'information de type référence d'entité non résolue 460
Unité d'information de type caractère 460
Unité d'information de type commentaire 460

Unité d'information de type déclaration de document 461
Unité d'information de type entité non analysée 461
Unité d'information de type notation 461
Unité d'information de type espace de noms 462

Glossaire **463**

Index **493**

Avant-propos

Une rencontre fortuite entre ses deux auteurs participe de la genèse de cet ouvrage. Le hasard a fait que nous soyons mis en présence un soir, l'un parlant de UML, l'autre de XML. Notre échange a mis au jour le vide qui existait entre la réalité et les discours de salon : contrairement aux idées reçues, utiliser UML pour concevoir des modèles XML n'est pas direct, et confrontées à la spécificité de XML, les anciennes méthodes de modélisation semblent échouer. Nous avons eu envie de clarifier la situation au moyen d'un ouvrage. L'évidence du problème s'est révélée encore plus dès que nous avons commencé à rédiger l'ouvrage ; nous n'imaginions pas à quel point notre intuition initiale était en deçà de la réalité.

Chacun dans sa spécialité a vu ses idées converger avec celles de l'autre ; cela s'est fait dans l'harmonie. C'est important. Comme Marcel Dassault, qui disait souvent qu'« un bon avion est un bel avion », nous pensons qu'un bon modèle est un beau modèle.

Faute de pouvoir tout connaître des applications de XML, nous étions certains d'au moins une chose : XML est bien plus qu'un nouveau format d'échange de données entre ordinateurs, bien plus qu'un alignement de balises et d'attributs. C'est un nouveau type de modèle de données qui, à ce titre, doit recevoir de nouvelles techniques de modélisation.

XML est pour les ordinateurs une nouvelle langue universelle qui va peut-être enfin leur permettre de s'ouvrir à la communication. Une langue qui serait toutefois aujourd'hui comme folle, parlée par mille personnes qui ne chercheraient ni à s'écouter, ni même à apprendre la langue de son interlocuteur. L'image est caricaturale mais l'impression générale laissée par l'avalanche de normes, standards et protocoles XML y confine parfois.

Les possibilités de la matière XML sont encore sous-estimées. Le *basus XMLius* ne fait que découvrir la puissance réelle de XML, qui va bien au-delà d'un simple format d'enregistrement. Voyez un peu : on exprime en XML des structures de données, des documents pourtant usuellement assimilés à des *données non structurées*, des descriptions de processus, des ressources informatiques physiques, mais également des concepts totalement abstraits, des langages de programmation, et la liste est encore longue… XML pourrait même envahir sous peu les couches basses des systèmes d'exploitation.

Une telle omniprésence ne peut qu'inciter les concepteurs d'applications à comprendre, bon gré mal gré, comment toutes ces architectures de données et d'information s'intègrent et cohabitent. XML permet de manipuler des espaces d'information à *n* dimensions. Au cœur d'applications de plus en plus interconnectées, les données XML, de par les enjeux économiques qu'elles représentent, devront être modélisées avec le plus grand soin.

Objectifs de ce livre

Comme tout langage, XML a ses règles et, comme toute création humaine, il n'expose sa puissance qu'à ceux qui le lui demandent.

Nous espérons répondre dans cet ouvrage à de nombreuses questions, à commencer par une explication du vocabulaire qui s'y trouve utilisé. Diverses influences font que des termes abscons sont trop souvent utilisés par les spécialistes : nous voulons les rendre aussi simples à comprendre que possible. Ces mots doivent, comme *frigidaire*, devenir courants. C'est ce souci de clarté, de simplicité, qui nous a guidés tout au long de cet ouvrage.

Au-delà de la théorie, nous mettons notre expérience à votre disposition : d'un côté, dix années de modélisation, et de l'autre, vingt ans de SGML et XML. Cette expérience nous autorise à formuler quelques conseils dans le choix des modèles et de la méthode. Ceux que nous présentons ont tous été confrontés à des projets réels.

Ce livre apporte des réponses aux questions suivantes : Quel est le bon modèle ? Comment concevoir un modèle ? Existe-t-il une méthode ? Comment mettre au point les schémas XML ? Comment mesurer la qualité d'un modèle ?

Un schéma XML est la traduction physique d'un modèle. Nous expliquons ici comment conduire la conception d'un tel modèle. Nous décrivons les différentes étapes qui permettent de passer du modèle théorique au schéma concret.

Peu à peu, la prise de conscience se fait que XML a un rôle à jouer bien plus grand que celui de simple format de fichier. Cet ouvrage est destiné tant à ceux qui en sont convaincus qu'à ceux qui découvrent la modélisation XML.

Ayant aboli les frontières traditionnelles de la donnée structurée, du document et de la transaction, XML s'est imposé *de facto* comme modèle universel dans la définition de schémas de stockage des données (XML Schema et Relax NG), d'échange (SOAP, UDDI, WSDL), d'application de règles métier (XSLT, XQuery, Schematron) et d'orchestration des processus (BPEL).

Au regard des couches logicielles qui composent les systèmes d'information apparaît naturellement le concept d'architectures de données. L'empilage des couches de données va confronter les concepteurs à un problème du type tour de Babel. On le voit déjà avec des applications telles que SOAP et JSP qui permettent de faire cohabiter dans un même fichier informatique plusieurs modèles XML, voire plusieurs langages de programmation.

C'est à ce stade que XML devient un outil de modélisation, passant du statut de simple descripteur de données (le schéma des données) à celui de chef d'orchestre (le schéma des processus qui le font vivre).

La question de la représentation des nouveaux systèmes n'a pas de réponse officielle. Il n'existe pas à ce jour de méthode de modélisation des systèmes qui aurait été spécialement étudiée pour XML. Cela viendra probablement un jour, mais le travail est difficile : XML est multiforme, la « chose » est plus complexe qu'il n'y paraît. Si l'information est la plus humaine des caractéristiques animales, sa gestion avec XML est la plus humaine des créations de l'informatique.

Nous cherchons désormais à représenter des flux d'informations qui, par nature, ont pour vocation à circuler, s'échanger, se transformer. XML est aujourd'hui la seule réponse que nous ayons face au besoin grandissant de gestion d'informations changeant d'état continuellement. Nous en tenons compte ici dès le chapitre 1.

Aussi, produire des schémas XML ne se limite pas à écrire des schémas statiques de données mais à représenter les différents états de l'information, de sa création à son obsolescence.

Concrètement, vous allez découvrir :

- une démarche de modélisation XML en utilisant UML ;
- les impacts de XML sur la conception des systèmes d'information ;
- les trucs et astuces de certains schémas XML.

À qui s'adresse cet ouvrage ?

« On comprend mieux nos ignorances d'avant » (témoignage d'un chef de projet ayant participé à un cours sur la modélisation XML).

Le livre s'adresse à ceux qui, pour concevoir des systèmes d'information, doivent recevoir, gérer et traiter des volumes importants de données XML. Aujourd'hui, ils se heurtent à des techniques de conception inadaptées au XML. Ce livre leur offre les explications dont ils ont besoin pour adapter leurs méthodes de conception aux cas auxquels ils sont confrontés.

Tous les informaticiens sont concernés par cet ouvrage.

Nous pensons tout d'abord aux étudiants en informatique qui ne manqueront pas d'être confrontés, dès leur entrée dans la vie professionnelle, à des projets faisant largement appel à XML.

Ensuite, nous pensons à ceux pour qui le XML était initialement dédié : les développeurs de sites de commerce électronique et, plus largement, ceux qui conçoivent et développent des applications de gestion de contenu.

Bien sûr, sont également concernés les développeurs de systèmes de gestion et production de documents techniques, juridiques ou autres.

Enfin, viennent les informaticiens du monde de la gestion qui évolue rapidement vers une informatique d'entreprise et, de ce fait, ouvre ses applications à la gestion des documents d'entreprise.

Sujets couverts par cet ouvrage

Ce livre fournit théorie et pratique :

- Les sept premiers chapitres sont autant d'étapes de la démarche de modélisation que nous expliquons.
- Les six suivants sont des exemples concrets de cas particuliers.

Dans notre **introduction**, nous revenons sur la dualité XML/UML avec des explications sur les concepts qui se cachent derrière les termes de modèle et de schéma.

Dans le **chapitre 1**, nous présentons une méthode de préconception d'application. Elle permet d'identifier, organiser et analyser les fonctions du futur système d'information. Cette méthode d'ébauchage se veut simple mais efficace. Elle offre aux chefs de projet qui la pratiquent la possibilité de rationaliser leur projet.

Dans le **chapitre 2**, nous analysons les modèles conceptuels des deux mondes XML et UML.

Dans le **chapitre 3**, nous expliquons comment passer du modèle conceptuel UML à un schéma XML. Nous y détaillons l'utilisation de la notation graphique du diagramme des classes de UML en vue de la production d'un schéma XML.

La réalisation d'un modèle physique constitue le **chapitre 4** où nous exposons les spécificités du stockage de documents XML. C'est ici qu'intervient la possibilité de prendre en compte les mécanismes de liaison de XML.

Dans le **chapitre 5**, qui traite de la séparation des données et des traitements, nous expliquons pourquoi l'approche dite par objets métier n'est pas bonne ; nous lui opposons en effet celle orientée service, plus conforme aux fondamentaux de XML.

Dans **le chapitre 6**, nous nous attachons à la problématique des variantes. À partir d'un modèle de base, il est fréquent de devoir développer des variantes. Ce chapitre expose tout ce qu'il faut savoir sur l'écriture de telles applications.

Le **chapitre 7** est fondamental. Comme nous l'avons vu en introduction de cet avant-propos, un fichier-programme peut contenir aujourd'hui jusqu'à quatre ou cinq langages de programmation différents ! Il faut savoir représenter ces programmes dans différents plans logiques pour s'assurer de leur cohérence.

Le **chapitre 8** est le premier d'une série traitant de cas réels. Les modèles choisis ont tous une originalité. Ils peuvent servir d'exemple. Dans ce chapitre, nous présentons les modèles modulaires qui sont articulés et peuvent être décomposés en petits morceaux.

Dans le **chapitre 9**, nous expliquons le rôle des éléments purement structuraux, qui sont aux schémas XML ce que les articulations sont au corps humain.

Le cas des modules d'information est traité au **chapitre 10**. Nous fournissons en particulier les règles de base qui gouvernent la conception de systèmes basés sur le principe des modules d'information.

Les métadonnées, qui font le pont entre la gestion du contenu et celle du contenant, sont présentées au **chapitre 11**. Les modèles RDF, Dublin Core, xHTML et S1000D sont présentés accompagnés de nombreux exemples concrets.

Les possibilités de liaison sont un point fort de XML. Sans les liens, le Web ne serait pas, à coup sûr, ce qu'il est devenu aujourd'hui. C'est un sujet important mais complexe. Nous revenons dans le **chapitre 12** sur les modèles ID/IDREF, XLink, XPointer, XTM, et expliquons en détail le principe des liens logiques.

Nous terminons notre voyage dans le monde des cas réels avec le **chapitre 13** qui aborde la question de la gestion des révisions. Garder la trace et l'historique des informations modifiées est aujourd'hui un impératif pour de nombreuses applications ; mieux vaut connaître les possibilités offertes par XML dans ce domaine.

Quelques règles de lecture de l'ouvrage

Les chapitres de l'ouvrage sont relativement indépendants les uns des autres. Vous pouvez donc les aborder librement. Chacun d'entre eux cerne un des aspects du sujet général de l'ouvrage, de la façon la plus didactique possible, étayée par moult exemples et illustrations.

L'ouvrage contient des extraits de schémas XML, des exemples de documents XML et des figures. Les tableaux sont également utilisés pour présenter certains cas de figure. La syntaxe des schémas XML fait que les codes sources sont longs et pénibles à lire. Parfois, nous avons préféré simplifier la lecture des schémas sources en les écrivant sous la forme DTD ou de représentations graphiques.

En ce qui concerne le vocabulaire, nous utilisons la terminologie suivante :

- Après avoir très longtemps combattu le mot « parseur » au profit de la seule expression consacrée d'« analyseur lexico-syntaxique », il apparaît que le mot « parseur », largement utilisé, s'impose. Il faut lui reconnaître l'avantage de la concision. Attention toutefois, ce mot est souvent galvaudé. Nous vous encourageons à vous référer au glossaire pour connaître sa définition exacte.
- Le terme de schéma pour XML est utilisé pour désigner un schéma conforme à la recommandation XML Schema du W3C. Pour la norme ISO Relax NG, nous parlerons de grammaire. Le terme DTD est utilisé pour parler d'un schéma conforme à la norme ISO 8879, à savoir SGML, ou la recommandation XML 1.0 du W3C. Pour éviter de devoir préciser le type de schéma dans les cas où cela n'est pas utile, nous parlerons simplement de schéma.
- Nous utilisons le terme d'« instance » pour désigner un ensemble d'information SGML ou XML valide par rapport à une DTD, et l'expression de « document XML » pour un ensemble d'information XML valide par rapport à un schéma XML.

Les premiers chapitres se lisent l'un à la suite de l'autre, mais ceux de la deuxième partie de l'ouvrage peuvent être lus dans n'importe quel ordre.

À propos des exemples

Nous nous interdisons de développer des théories sur des exemples imaginaires. Il existe un gouffre entre la réalité et les sempiternels exemples de bons de commandes, bibliothèques virtuelles et autres recettes de pizza *al dente* dont la littérature technique nous abreuve. Proposer un ouvrage sur la modélisation XML qui ne serait pas l'image la plus précise de la réalité serait une démarche *de facto* pervertie. Toutefois,

nous sommes parfois obligés de faire l'impasse sur ce principe et d'avoir recours à des exemples figurés. Ils sont alors la plupart du temps simplement détournés de cas réels.

Notre politique sur cette question est nette. En accord avec Kant qui s'attache à la manière dont les chercheurs font l'acquisition de leurs connaissances scientifiques, nous pensons que le véritable savoir ne s'acquiert que par la confrontation de postulats à la réalité.

« Avec XML, je sais travailler pour les générations futures,
c'est du développement durable ! »

Jean-Jacques Thomasson

« XML ouvre de nouveaux horizons encore inexplorés
à l'application des techniques de modélisation »

Antoine Lonjon

Les auteurs

Jean-Jacques Thomasson a suivi depuis 1984 toute l'évolution des langages de balisage, de SGML à XML. Il dirige depuis vingt ans des équipes spécialisées dans le développement de systèmes de gestion de l'information. Riche de cette longue expérience de la mise en œuvre de XML au sein des systèmes documentaires, il milite chaque fois qu'il le peut en faveur d'une nouvelle approche de la modélisation des données, seule capable selon lui, de faire évoluer de manière décisive la relation homme-machine ainsi que la puissance des applications informatiques. Précédemment à cet ouvrage, il a traduit plusieurs recommandations du W3C (XSLT, DOM, Xpath, XML Schema), les normes WebDAV de l'IETF et XTM de XML Topic Map, le livre d'Eric van der Vlist sur XML Schema paru aux éditions O'Reilly. Il est également l'auteur de *Schémas XML* paru aux éditions Eyrolles en 2002, ainsi que de cinq articles édités depuis 2004 par la revue *Informatique Professionnelle* du Gartner Group. Contact : jjthomasson@free.fr.

Antoine Lonjon a plus de 15 ans d'expérience dans le domaine de la modélisation appliquée à l'analyse des systèmes d'information et des métiers de l'entreprise. Il a participé à de nombreux projets internationaux de standardisation des techniques de modélisation, dont le projet des *Core Component* de ebXML et le projet *UML 2.0* à l'OMG (Object Management Group). Antoine Lonjon a aussi contribué à la conception des outils de modélisation de la suite logicielle de MEGA International dont il est est aujourd'hui directeur de l'offre.

Remerciements

Nous tenons à remercier tous ceux qui, de près ou de loin, ont croisé la route de ce livre et accepté de nous accompagner par la discussion, l'écriture ou la relecture :

- Libero Maesano, le bénédictin de notre ouvrage et grand maître des services Web depuis la parution aux éditions Eyrolles de son ouvrage de référence sur le sujet. Sa vision sur notre travail était fondamentale : les architectures orientées service (SOA) et les modèles de données XML sont comme des prises mâles et femelles.
- Jean Delahousse et son équipe de Mondeca prônent l'utilisation du modèle XTM (XML Topic Map) pour la gestion des connaissances. Michel Biezunski a contribué de manière importante à la rédaction de la norme ISO 13250 spécifiant les Topic Maps dont est hérité le modèle XTM. Tous deux sont des amis qui ont accepté de relire les parties de l'ouvrage concernant ces sujets.
- Michel Doméon, de la société Dassault-Aviation, qui m'a soutenu depuis des années dans mes travaux et a apporté son savoir sur la norme S1000D. Il fait référence sur la question de la modularisation de l'information et intervient au plus haut niveau des instances décisionnaires concernant les normes et modèles XML applicables au monde de l'aéronautique et de la défense.
- Bruno Estrade, qui a travaillé depuis plusieurs années sur les systèmes de bases de données mêlant relationnel et XML, a joué un rôle important dans la rédaction du chapitre dédié au stockage (chapitre 4) en multipliant les tests et en confrontant la théorie à la réalité. Ce chapitre, commencé par Julien Viet, n'aurait pu être terminé sans son précieux concours.
- Guillaume Lebleu, de la société Brixlogic, a apporté son regard et validé nos propos sur les modèles orientés données utilisés dans les milieux de la banque et de la finance. Brixlogic développe une offre de produits qui comprennent la logique métier de ces modèles XML et permettent donc aux utilisateurs de se concentrer sur leur seule problématique métier.
- Stéphane Mariel nous a autorisé à reproduire à l'identique l'application PiloteWeb qu'il avait conçue et développée pour son propre ouvrage paru aux éditions Eyrolles : *PostgreSQL* dans la collection des *Cahiers du programmeur*. Qu'il en soit ici remercié, ce cas d'école est intéressant à plus d'un titre.
- Nicolas Rabaté, qui a toujours cru en nos travaux (depuis près de vingt ans !), nous a soutenu, et a relu le chapitre et la théorie sur les éléments purement structurels à une époque où le texte était loin d'être abouti...
- Arthur Ramiandrisoa, architecte de la société SQLI, qui, il y a plusieurs années déjà, fut l'un des premiers à comprendre le parti qu'il pourrait tirer des infrastructures XML, notamment grâce aux bases de données natives XML. Nous regret-

tons encore aujourd'hui que son nom ne vienne pas compléter les deux nôtres au générique de cet ouvrage.

- Julien Viet, jeune chercheur du NIST à l'époque où nous l'avons connu. Julien a fait un travail important sur la problématique de stockage de données XML dans des bases relationnelles avant de quitter le projet pour devenir chef développeur de JBoss.

Introduction générale

« Un document est un ensemble d'informations sélectionnées,
assemblées et organisées pour permettre à un utilisateur
donné de remplir une mission donnée »

Référence : « Guide pour la réalisation de spécifications
d'élaboration de documentation structurée » –
juin 1991 – Direction Générale de l'Armement.

Cette définition de 1991 est plus que jamais d'actualité. Initialement destinée aux seuls documents sur papier, elle convient aux fichiers XML, qu'ils représentent des documents papier, électroniques, ou des données. En particulier, on comprend que cette définition convienne parfaitement bien aux fragments XML échangés entre systèmes par le biais des services web : seul le mot « utilisateur » devrait maintenant être remplacé par l'expression « utilisateur ou système ». Désormais, tout fichier XML bien formé est appelé document, plus précisément *document XML*. On ne fait plus la différence entre données et documents, la frontière étanche qui existait entre ces deux mondes est abolie et l'on devrait même bientôt simplement utiliser le mot *Infoset*, ou ensemble d'information.

Pour les informaticiens, il s'agit d'un changement radical de perspective, d'une révolution importante qui bouleverse jusqu'à la conception des applications.

> **VOCABULAIRE Infoset et document XML**
>
> Infoset signifie littéralement *Information Set*, soit en français ensemble d'information. La recommandation du W3C, *XML Information Set*, datée du 24 octobre 2001, donne une définition abstraite de tous les types d'objets dont la présence est autorisée dans les documents XML. *Ensemble d'information* est donc la désignation la plus générale que l'on puisse donner à un document XML.
> Est baptisé document XML tout fichier contenant des données balisées conformément aux règles définies par cette recommandation du W3C. Le mot document n'est donc pas ici à prendre dans son sens commun.

Nous donnons les clés de ces changements dans cet ouvrage, en termes sobres le plus souvent, et en ayant épuré notre vocabulaire des effets de mode néfastes à la réflexion. Nous cherchons à présenter des invariants, pas des courants passagers.

Modèles et schémas doivent, par définition, garantir la stabilité dans le temps : les applications informatiques concernées par cet ouvrage sont faites pour durer aussi longtemps que les organisations qu'elles servent.

Graver les choses dans le marbre de la modélisation ne revient toutefois pas à rigidifier. Et même bien au contraire, modéliser est synonyme de prévision et anticipation. Ouvrir largement les systèmes informatiques est de nos jours un impératif : extensibilité et adaptabilité sont les maîtres mots de l'intégration.

Sur ce point, XML est précisément une bonne réponse au défi technologique né de la mondialisation des organisations et du commerce. Ne devant en aucun cas être réduit au simple rôle de technique de balisage, XML est un véritable type de structuration de l'information qui vaut modèle. C'est le candidat idéal des systèmes d'information de demain : au niveau de la mémoire où l'on retrouve les données, des liens qui forment l'information, et enfin des traitements qui rythment son fonctionnement.

La mise en musique de ces différentes facettes de XML ne peut se faire sans rationalisation du problème de modélisation des systèmes d'information ; cela est le projet même de cet ouvrage.

Pourquoi des modèles ?

« S'il te plaît, dessine-moi un mouton ! »

Le petit prince – Saint-Exupéry

Les modèles représentent une version simplifiée mais synthétique de la réalité. Le premier objectif d'un modèle est de faire progresser la compréhension d'un domaine étudié. Le modèle est le plan ou la carte qui permet aux acteurs d'un projet de communiquer et échanger sur la base d'une représentation commune, et acceptée de tous, des idées du projet.

Un modèle représente ce qu'il faut reproduire. Le mot est ambigu. L'acte de reproduction peut recouvrir l'idée d'une reproduction à l'identique. Aussi, les modèles peuvent-ils avoir une seconde fonction qui s'apparente à celle d'un moule. Cela est vrai tant pour le modèle du peintre et le moule en cire du fondeur que pour le développeur qui doit suivre un modèle décrivant des données et fonctions. Toutefois, n'oublions pas de penser tout simplement à la nature humaine pour qui la reproduction n'est pas la fabrication d'une copie à l'identique !

Ainsi, un modèle peut tout aussi bien être une chose du monde réel (le modèle d'un peintre est par exemple un être vivant ou des fruits sur un plateau…) que pure création (les plans d'une nouvelle construction).

Il est important de distinguer ces deux modes d'utilisation des modèles. On peut toujours utiliser un modèle pour représenter le domaine étudié, mais on ne s'en sert pas systématiquement comme moule. Dans le contexte de l'analyse des systèmes d'information, cette distinction est cruciale et ne pas la faire est source de nombreuses incompréhensions. Par exemple, on peut concevoir un modèle de données des contrats d'assurance pour comprendre le monde de l'assurance – les contrats, les avenants, la réglementation – ou pour produire l'organisation exacte de la base de données correspondante. Dans les deux cas, le niveau de détail et de préoccupation sera différent. De même, on peut chercher à décrire les consignes de production des manuels de formation des nouveaux arrivants ou les processus très complexes des opérations de traitement des dossiers de crédit immobilier.

La modélisation est une technique qui peut s'adresser à tous les pans de l'activité humaine. Dans cet ouvrage, le domaine étudié est celui des systèmes d'information, et plus particulièrement celui de la représentation des données. On peut donc y repérer deux niveaux de préoccupation : l'un est celui de la modélisation elle-même, l'autre de ses applications au domaine du traitement de l'information.

La question des modèles n'est pas spécifique à XML : « *Le choix est toujours le même. Soit vous rendez votre modèle plus complexe et plus fidèle à la réalité, soit vous le rendez plus simple et plus facile à manipuler. Seul un scientifique naïf peut penser que le modèle parfait est celui qui représente parfaitement la réalité. Un tel modèle aurait les mêmes inconvénients qu'un plan aussi grand et détaillé que la ville qu'il représente, un plan indiquant chaque parc, chaque rue, chaque bâtiment, chaque arbre, chaque poteau, chaque habitant. Si un tel plan était possible, sa précision irait à l'encontre de sa destination première : généraliser et résumer. Les cartographes soulignent ces caractéristiques auprès de leurs clients. Quelles que soient leurs fonctions, les cartes et les modèles doivent tout autant simplifier le monde que le reproduire.* » Extrait de *La Théorie du chaos*.

📖 *La Théorie du chaos – Vers une nouvelle science*, James Gleick, éditions Flammarion, traduction de 1989, ISBN 2-08-081219-X.

UML et XML : un duo gagnant

La tentation commune à toute activité de modélisation est la recherche de l'universalité, du modèle à tout faire : dès lors que l'on a un marteau, tous les problèmes doivent avoir une tête de clou. Cette attitude a deux conséquences majeures sur l'approche par les modèles :

- En étendant indéfiniment leur champ d'application, elle rend impossible la définition d'un périmètre clair de ce que décrivent les modèles en question. On ne sait plus de quoi parle le modèle puisqu'il veut couvrir le Tout.
- Elle conduit à confondre les modèles avec la réalité : la carte n'est pas le territoire. Une telle confusion peut entraîner bien des désillusions lors du passage à la pratique.

On retrouve souvent cette attitude dans l'utilisation d'UML dont on confond le « U » de *Unified* avec celui de *Universal*. Certains modèles XML n'échappent pas non plus à cette tentation. Complet, le modèle DocBook cherche à couvrir un périmètre trop vaste et trop complexe (écrit en XML Schema, le modèle fait 8 500 lignes) pour être véritablement efficace. Il en est réduit à servir de catalogue à tout faire, jamais ignoré, jamais totalement mis en œuvre, soit exactement le contraire des souhaits initiaux de ses concepteurs.

La tendance à l'élargissement est d'autant plus marquée dans l'informatique que nous venons d'un monde centralisé où données, traitements et communications étaient concentrés sur un ensemble d'applications régentées par un seul et même environnement.

En explosant, les capacités de stockage, de communication et de vitesse de calcul des processeurs ont ouvert un vaste champ de possibilités que les modèles doivent permettre de contrôler.

À époque révolue modèles révolus !

En complément à ses performances physiques, XML joue, sur le plan de la conception des données, le rôle d'un outil révolutionnaire. La capacité des données XML à porter leur propre définition indépendamment de tout programme est un pas décisif vers l'autonomie des données.

Le spectre de la modélisation des données depuis l'expression des besoins fonctionnels jusqu'à la solution opérationnelle nécessite de faire appel à trois types de modèles de données :

- Les modèles conceptuels expriment le vocabulaire et les concepts que l'on souhaite traduire sous la forme d'une application. Ils permettent d'échanger des définitions claires et servent à fixer les limites des concepts. La plupart du temps, les modèles conceptuels sont exprimés sous forme graphique : *un dessin vaut mieux qu'une longue explication*. Dans notre ouvrage, les modèles conceptuels sont représentés par des diagrammes de classe UML.

- Les modèles logiques décrivent les structures de données correspondant aux concepts. Pour XML, cela se traduit par la structure arborescente des éléments. Le modèle logique sera représenté pour XML par une DTD ou un schéma XML, et pour UML par un diagramme de classe adapté à XML.
- Les modèles physiques expriment les choix retenus pour l'implémentation technique des données : il se peut que les modèles logiques soient décomposés, que le modèle de stockage diffère du modèle logique, que des liens différents de ceux du modèle conceptuel soient établis. Les documents XML eux-mêmes peuvent être stockés selon différentes techniques (systèmes de fichiers, bases de données…).

Alors que XML ouvre de nouvelles possibilités, les méthodes de modélisation classiques n'en tiennent pour autant pas compte. Elles n'intègrent pas les spécificités de XML dans leurs considérations. Cet ouvrage apporte, en ce qui concerne UML, une *méthode* pour utiliser UML afin de concevoir des schémas XML.

UML et les différentes vues d'un système d'information

Les systèmes d'information ont vocation à communiquer des données après les avoir dé-stockées, transformées et transmises. Ainsi, leur analyse montre qu'ils s'articulent autour de trois composantes : le stockage, le traitement et la communication. En d'autres termes, on ne peut parler d'information qu'à partir du moment où des données sont sélectionnées, assemblées, mises en forme et transmises. Voilà une périphrase parfaite de la définition écrite en début de chapitre.

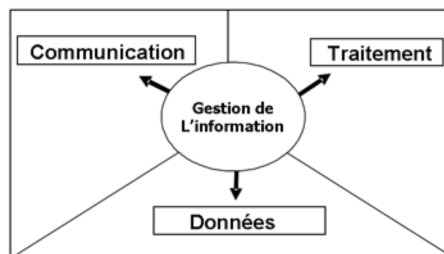


Figure I–1
Les trois composantes d'un système d'information

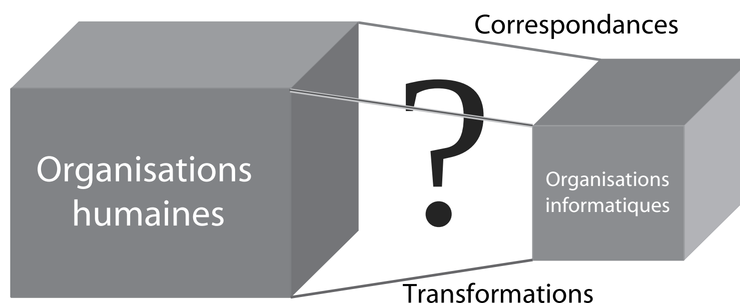
Le sujet qui nous intéresse au premier chef dans cet ouvrage est celui de la définition et de la représentation de la structure des données. La figure I-1 montre que cette composante est à la fois proche et indissociable des deux autres. Pour être transmise, une information doit être extraite d'un système et transformée ; les traitements alors effectués ont un impact sur sa structure, et réciproquement. Une information mal structurée rendra difficile une exécution normale des traitements et toute structure induira un certain type de traitement.

À cela s'ajoutent deux points de vue sur l'information : le premier est relatif aux processus informatiques qui la traitent et le second à l'organisation humaine qui l'utilise.

Les deux sont difficiles à faire coïncider. Dans la pratique, faire évoluer une organisation humaine peut s'avérer plus compliqué que ce ne l'est pour un système informatique… et *vice versa*, selon les cas.

Ainsi, la modélisation d'un système d'information se ramène bien souvent à faire coller différentes pièces d'un puzzle animées de forces antagonistes. La figure I-2 représente ce problème consistant à faire tenir une grande boîte dans une petite.

Figure I–2
Le système informatique est forcément une vue réductrice d'une organisation humaine



Quand l'organisation humaine touche des milliers voire des millions de personnes avec des ramifications dans des centaines de pays, la modélisation du système s'impose. En permettant de garder la maîtrise de la représentation du système, elle ouvre la voie à la mise au point d'évolutions, d'extensions, de modifications.

La modélisation est alors perçue comme l'outillage de base du pilotage des projets informatiques. Dans ce contexte, ses domaines d'application sont aussi variés que l'analyse des exigences, de l'organisation, des objectifs, des fonctions attendues du système et de son architecture. L'ensemble de ces disciplines est généralement appelé *architecture d'entreprise*. Il n'est pas dans la vocation d'UML de couvrir la totalité de ce spectre. Malgré ses mécanismes d'extension, UML s'adresse avant toute autre chose au domaine de la conception orientée objet. Dans cet ouvrage, UML est utilisé pour l'analyse des données et la conception du modèle de classe, toutes deux au cœur de notre sujet. S'il existe de multiples manières d'exploiter XML pour représenter des données, le modèle de classe d'UML permet de disposer d'une vue conceptuelle unifiée, et sa notation graphique est un outil essentiel des phases d'analyse.

Le modèle de classe ne représente toutefois pas la dynamique d'un système d'information. Pour cela, nous ferons appel à des techniques de modélisation proches de celles utilisées dans l'urbanisation des systèmes d'information et les architectures de services.

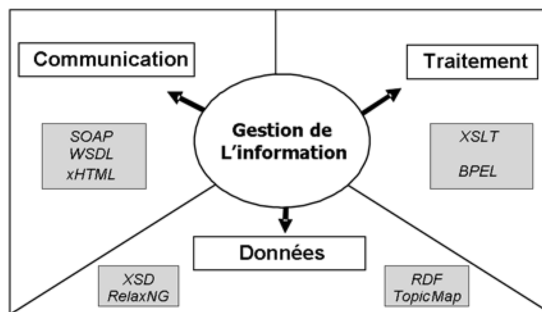
XML au cœur des systèmes d'information

En structurant les données et l'information de manière universelle, le langage XML s'applique à tous les domaines techniques relatifs à la gestion de l'information. De ce fait, XML n'est pas, et de loin, cantonné à son seul stockage, comme c'est le cas du SQL avec le relationnel. XML est omniprésent.

Dans la figure I-3, nous indiquons que les applications de XML couvrent la totalité des trois composantes de base d'un système d'information :

- Le stockage des données dans des bases de données XML (XML Schema ou Relax NG) en définissent les types, la classification revenant à RDF ou Topic Map.
- La transformation des données et l'orchestration des processus sont des traitements exprimés en XML grâce à des recommandations telles que XSLT et BPEL.
- La communication des données peut, quant à elle, être assurée par xHTML pour l'affichage, et la trilogie des services web : Soap, WSDL et UDDI.

Figure I–3
L'applicabilité quasi universelle de XML



Même si chaque composant du système conserve ses spécificités et prérogatives, un principe commun de structuration est mis en œuvre : XML. Cela va permettre de concevoir des systèmes d'information beaucoup plus performants que ceux en vigueur actuellement.

Apports de XML à la modélisation

Cette section revient sur les changements parfois importants induits par XML. Ces nouveautés vont le plus souvent dans le bon sens : les tâches d'élaboration et de maintenance des modèles logiques et physiques sont rendues plus simples, rapides et efficaces.

Il y a plusieurs catégories de nouveautés :

- Les unes sont intrinsèques au méta-modèle XML.
- Les autres sont consécutives au travail normatif réalisé par l'ensemble de la communauté XML, et en particulier le W3C.

> **BLABLAWARE La communauté XML**
>
> Nébuleuse d'experts répartis dans le monde constituée depuis une vingtaine d'années à partir des promoteurs historiques de SGML, puis de XML. Une force particulière les unit qui résulte de leurs convictions quant à la révolution que représente XML et des difficultés passées à se faire entendre.

> **RÉFÉRENCE Le W3C**
>
> Le W3C, ou World Wide Web Consortium, est une association fondée par l'inventeur du Web, Tim Berners-Lee, récemment anobli par la reine d'Angleterre en reconnaissance des services rendus à la communauté des hommes.

L'universalité de XML

Cette caractéristique est sans conteste l'une des plus importantes de XML.

L'universalité de XML provient de l'adoption d'une syntaxe simple et puissante qui représente un modèle des plus génériques de représentation des formes : une hiérarchie d'éléments, leurs attributs et leur contenu textuel. C'est un peu comme si toutes les langues du monde utilisaient les mêmes constructions de phrases !

Cette forme commune à tous les documents XML n'a pas été conçue, *a priori*, pour représenter les seules informations d'une base de données : le formalisme XML s'applique à une grande variété d'applications informatiques. Il n'y a même, semble-t-il, aucune limite.

En cela, le langage XML est évidemment très différent du « modèle » relationnel, confiné aux seules applications de gestion. Par exemple, il est difficile de représenter un fichier CAO (conception assistée par ordinateur) sous forme relationnelle. Il en va de même des documents techniques, pour lesquels le relationnel ne sait pas, sans contorsion, gérer l'ordre d'apparition des paragraphes ni les modèles de contenu mixte. On pourrait encore citer le cas de la programmation des processus… Tout cela est difficile en relationnel mais simple en XML.

Il est fini le temps où les applications utilisaient des formats propriétaires qui ne pouvaient facilement être échangés avec d'autres applications. Les éditeurs produisent désormais des logiciels permettant d'échanger les données avec d'autres logiciels *via* un format XML. Données et traitements sont clairement séparés.

L'interopérabilité des documents XML

Grâce à leur syntaxe unique et universelle, les documents XML sont facilement transportables entre ordinateurs. Même en considérant qu'il existe plusieurs vocabulaires, ou dialectes, les uns et les autres utilisent le même format, la même syntaxe, la même grammaire. Aussi, quand les modèles sont eux-mêmes écrits en XML, on peut dire que les ordinateurs parlent de concert la même langue.

De plus, documents et modèles XML peuvent être lus par les humains sans aucune perte d'information. Les développeurs du monde entier parlent, eux aussi, la même langue.

Dans ces conditions, les erreurs d'interprétation sont d'autant diminuées. Tout modèle XML peut être précisément discuté et évalué sur le fond.

Cela est important : en diminuant le nombre de transpositions entre représentations logiques et physiques, XML diminue de manière significative le temps de conception, analyse et pré-étude des nouvelles applications.

Et bien plus, l'interchangeabilité entre ordinateur s'accroît avec l'arrivée de XML dans les couches basses des systèmes d'exploitation et de communication. Il existe désormais des infrastructures matérielles de communication qui routent les messages en fonction du contenu XML de leur en-tête ou les transforment à une cadence industrielle. Il s'agit des « hardware XML ».

L'indépendance entre modèles et données

Toutes les choses ont une forme. C'est même grâce à cette représentation que nous communiquons sur les choses nous entourant. Quatre pieds, un plan horizontal, un dossier : c'est une chaise.

On peut parler de la forme particulière d'une chaise ou de la forme commune à un ensemble de chaises. On dira alors qu'il s'agit d'un modèle de chaises. Cette distinction est cruciale. Elle permet de parler d'une chose particulière indépendamment de son modèle. Il en est de même en XML. Il est possible d'écrire un document sans avoir recours à un schéma. Pour reprendre notre exemple précédent, il est possible de parler de « cette chaise » indépendamment du plan de toutes les chaises qui lui ressemblent.

L'informatique a été dès l'origine conçue pour capturer l'information de manière reproductible. Cela a conduit à mettre l'accent sur les modèles en tant que moules. Ainsi, dans le monde des bases de données relationnelles, si l'on veut parler d'une chaise, il faudra d'abord créer le schéma relationnel « chaise ». Ce dernier devra comprendre toutes les caractéristiques possibles de toutes les chaises que l'on veut décrire. En d'autres termes, on ne peut pas parler d'une chose particulière sans la faire entrer dans un moule. Il faut donc avoir prévu tous les cas possibles nécessaires à la description. Si par malheur le moule ne correspond plus, il faut le modifier.

C'est exactement ce qui se produit dans les systèmes informatiques actuels, ce qui les rend très rigides. Pour intégrer un nouveau type de chaise, il faut modifier la structure de la base de données. Il se produit la même chose avec les classes Java ou .Net.

Avec XML, une véritable déconnexion s'opère entre le monde des choses – les documents XML – et celui des moules – les schémas. On peut écrire un document XML sans avoir jamais recours à un schéma ou une DTD. On peut construire un schéma par la suite pour valider le document. Des milliers de schémas pourraient valider le même document XML. Il existe même plusieurs langages d'écriture de schémas correspondant à différents types de validation. La déconnexion offre un facteur de souplesse inédit qui aura de nombreux impacts sur le mode d'organisation des données, sur leur cycle de vie, et donc sur les traitements qui y sont associés.

La forme des modèles XML

VOCABULAIRE Forme

Nous avons choisi de parler de la forme des documents et des modèles XML en référence à « document XML bien formé », traduction de « XML well-formed document », expression qui signifie que la syntaxe XML est correctement mise en œuvre.

En XML, les modèles sont écrits à l'aide d'une DTD ou d'un schéma. Ces langages ont été conçus de manière à :

- être produits à partir d'un simple éditeur ASCII ;
- en permettre une mise en œuvre immédiate, sans recours à aucune transformation ;
- dissocier le modèle conceptuel des données de leur représentation structurée sous forme XML (nous verrons que les deux représentations sont largement complémentaires : l'une représentant la sémantique des données pour une application et l'autre leur structure pratique) ;
- ne pas mêler données et traitements.

VOCABULAIRE Données et traitements

L'un des problèmes à résoudre dans un système d'information est celui de la frontière entre données et traitements : nous savons que les traitements sont consommateurs de données et plusieurs traitements peuvent consommer les mêmes données. La question est alors de savoir s'il est possible de trouver des structures de données communes à tous ces traitements. Nous verrons dans cet ouvrage que XML permet de repousser cette frontière.

XML Schema, Relax NG et Schematron sont l'aboutissement de la volonté d'appliquer la forme générale de XML aussi bien aux documents qu'à leurs modèles. Ces nouveaux langages de modélisation permettent d'écrire les modèles XML… en XML.

Tous les programmes qui s'appliquaient au traitement des documents peuvent s'appliquer aux modèles. Par exemple, on peut concevoir une feuille de style XSLT pour transformer des modèles afin de les afficher en HTML, SVG, ou encore générer une documentation RTF. On peut exploiter la forme d'un document XML (c'est-à-dire le balisage) sans en exploiter le fond.

C'est ainsi que XML n'impose finalement ni une méthode d'écriture de schéma ni même une syntaxe de conception. Libre à chacun de trouver la forme d'écriture idéale à sa conception. Par exemple, la société américaine Brixlogic a développé des éditeurs de processus destinés aux métiers de la banque et de la finance s'appuyant sur les modèles métier que sont FpML, OFX, IFX et SwiftML. Cette société propose ainsi à ses clients des outils de modélisation parfaitement adaptés à leur métier.

Les apports du travail de normalisation

Une deuxième série d'éléments positifs concernant XML réside dans le nombre de normes et, par voie de conséquence, d'applications mettant en œuvre le langage.

Cela fait une très grande différence avec ce que fut la situation avec SGML ou même HTML :

- SGML avait une richesse d'expression de modèles sensiblement équivalente à XML, mais mettait l'accent sur la possibilité de faire varier les syntaxes concrètes : autant dire, les règles de base du langage. Cela lui fut fatal : avec l'arrivée d'Unicode, cette caractéristique technique est devenue à la fois trop lourde à prendre en compte et d'un intérêt plus que limité.
- HTML fut initialement perçu comme une solution magique, mais dont les experts pointèrent très vite les lacunes : balisage non régulier et sémantique pauvre, peu structurante et non extensible. Cela lui fut également fatal : HTML resta confiné au monde de l'affichage de pages web et fut remplacé par XML pour le reste.

Les normes développées autour de XML forment une galaxie de langages indépendants de tout vendeur de logiciel, ce qui rend le choix de XML encore plus attractif aux yeux des décideurs.

On peut regrouper ces normes en trois catégories :

- La première est celle des normes de référence qui constituent ce que l'on pourrait dénommer *l'infrastructure des modèles XML*. C'est par exemple les normes XML Namespace, Xlink, XML Schema. Elles donnent des capacités supplémentaires à XML pour exprimer différents types de données (des liens, des métadonnées, des

graphiques, etc.). On remarquera que ce groupe peut se subdiviser en deux : d'un côté, le groupe des normes ISO (Relax NG, Schematron), et de l'autre celui des recommandations du W3C.

- La deuxième regroupe les normes qui utilisent XML pour manipuler des données XML. Ces normes peuvent être apparentées à des langages de programmation. Il s'agit par exemple des normes WSDL pour les services web, BPEL pour les processus métier, XSLT pour les transformations de données, XQuery et XUpdate pour les opérations de stockage et dé-stockage des données, etc.
- La troisième catégorie est celle des normes de type métier qui s'appuient sur les précédentes pour établir des standards sectoriels. Elles sont le fruit du travail de groupes de réflexion qui étudient les modèles les mieux adaptés à leurs métiers : l'aéronautique, la banque, l'assurance, la santé…

Tableau I–1 Première catégorie, exemples de normes de référence

Les DTD	Langage d'écriture de schémas pour SGML et XML.
PSVI	Ensemble d'informations contenant le document XML et le schéma qui le valide.
RDF	Définition de modèles de métadonnées.
Relax NG	Langage d'écriture de schémas XML normalisé par l'ISO.
Schematron	Langage d'écriture de schémas XML permettant de spécifier des règles métier.
SVG	Vocabulaire XML de représentation d'illustrations en 2D.
VRML ou X3D	Vocabulaires XML de représentation d'objets graphiques en 3D.
Xinclude	Inclusions de fichiers XML entre eux.
Xlink	Langage de liaison.
XML 1.0 et 1.1	Langage de structuration de l'information.
XML Infoset	Spécification des unités d'information autorisées dans un document XML.
XML Namespace	Syntaxe et règles applicables au nom des modèles XML.
XML Schema	Langage d'écriture de schémas XML du W3C.
Xpath	Langage d'adressage d'unités d'information depuis l'extérieur d'un document XML.
Xpointer	Langage de ciblage d'unités d'information depuis l'intérieur d'un document XML.
XTM	Vocabulaire XML de description de liens thématiques entre ressources.

Tableau I–2 Deuxième catégorie, exemples de normes d'échange et de traitement

BPEL	Langage de spécification de processus métier.
SOAP, UDDI et WSDL	Langages des services web.
UBL	Modèle de documents de type commerciaux (commandes, facturation…)

Tableau I–2 Deuxième catégorie, exemples de normes d'échange et de traitement (suite)

XMI	Vocabulaire de description de modèles UML.
XSLT	Langage de transformation de documents XML.
Xquery et Xupdate	Langage de requête de bases de données XML.
WML	Vocabulaire XML dédié aux applications WAP du téléphone portable.
xHTML	Format d'affichage de documents dans un navigateur.

Tableau I–3 Troisième catégorie, exemples de normes sectorielles

Aéronautique	ATA 2200, S1000D, AECMA 2000M
Automobile	J2008
Audiovisuel	MPEG 4 et MPEG 7
Banque	SwiftML, OFX, IFX, FpML
Commerce	EbXML, UBL
Santé	HL7
Formation	SCORM

Un facteur d'adaptation des applications

L'expansion géographique et fonctionnelle des applications informatiques a une conséquence étrange : ces dernières deviennent théoriquement susceptibles d'évoluer en permanence.

L'augmentation du nombre d'utilisateurs, des cas particuliers à traiter, des systèmes à interconnecter fait que, à l'échelle de la planète, l'environnement informatique des applications change constamment.

En conséquence, il est impératif, d'une part de banaliser les couches système et matériel, et d'autre part de rendre plus flexibles les couches métier… parmi lesquelles on va trouver les données.

XML, encore lui, est pressenti comme bon candidat à l'élection de « format pivot universel des systèmes d'information ».

L'organisation OASIS souligne bien l'importance de ce point dans l'énoncé des avantages attendus de UBL, un vocabulaire XML qu'elle propose pour faciliter les échanges commerciaux :

« Nous attendons de ce standard les avantages suivants :

- *Un coût plus faible d'intégration, tant à l'extérieur qu'à l'intérieur des entreprises, grâce à la réutilisation de structures de données communes.*

- *Une diminution du prix des logiciels parce que développer un logiciel devant respecter un balisage prédéfini est bien plus facile qu'écrire un logiciel reconnaissant un nombre illimité de balises.*
- *Un apprentissage plus facile, parce que les utilisateurs ne doivent maîtriser qu'une seule bibliothèque d'éléments.*
- *Une plus grande facilité d'accès à la technologie pour les PME.*
- *Des formations standardisées, permettant d'augmenter le nombre de travailleurs formés.*
- *Le développement d'un pool universel de compétences en intégration de systèmes.*

On attend aussi de l'adoption de UBL qu'elle encourage la création de programmes peu coûteux d'import/export des données et que ce standard fournisse une syntaxe universellement connue et reconnue, et qu'elle soit légalement associée aux documents commerciaux. »

L'adressage

Les capacités d'adressage sont multiples. Avec XML, on peut facilement :

- Identifier une donnée XML : à cette fin, des mécanismes d'identification permettent de contrôler l'unicité d'une information. Les types ID/IDREF et les mécanismes `unique`, `key` (clé) et `keyref` (référence de clé) de XML Schema du W3C sont de bons outils d'identification et de contrôle d'unicité.
- Cibler une ressource à partir d'un contexte totalement étranger à XML. Les ressources cibles peuvent être identifiées par des descriptions de chemin d'accès absolues ou calculées. Les normes Xpath et Xpointer du W3C servent à cela.
- Relier les données les unes aux autres. Tous les types de liens sont prévus en XML ; depuis les simples URN (composés des URL et des URI) jusqu'aux liens complexes Xlink ou sémantique (XTM).
- Décrire une donnée en la ciblant de manière indirecte ou en utilisant des mécanismes de rebond. Toutes ces techniques sont mises en œuvre par les normes RDF et XTM (XML Topic Map), respectivement faites pour décrire des données et gérer des liens entre elles.

Si l'adressage est facile et offre un large éventail de possibilités, il en découle alors naturellement que le stockage sera, lui aussi, grandement facilité. En fait, l'un va avec l'autre, à la manière d'un miroir réfléchissant : c'est aussi parce que XML a de bonnes capacités de stockage que le ciblage des données est facile.

Le stockage

Bien qu'il soit impossible d'éclater un document XML en une série de répertoires et sous-fichiers, XML est une extension naturelle d'un système de fichiers.

Ainsi, le premier type de navigation que fait apparaître le modèle XML est la relation parents-enfants identique à celle utilisée par le système de répertoires et fichiers de n'importe quel système d'exploitation.

Une base de données XML donne aussi l'impression aux utilisateurs d'être un système de répertoires et fichiers. Toutefois, à la différence d'un système d'exploitation classique, on passe d'un fichier à son contenu de manière continue. Les traditionnelles frontières du fichier n'existent pour ainsi dire plus. La figure I-4 illustre ce propos. On y voit des répertoires ouverts, ainsi qu'un fichier et, sous lui, une partie de son contenu arborescent.

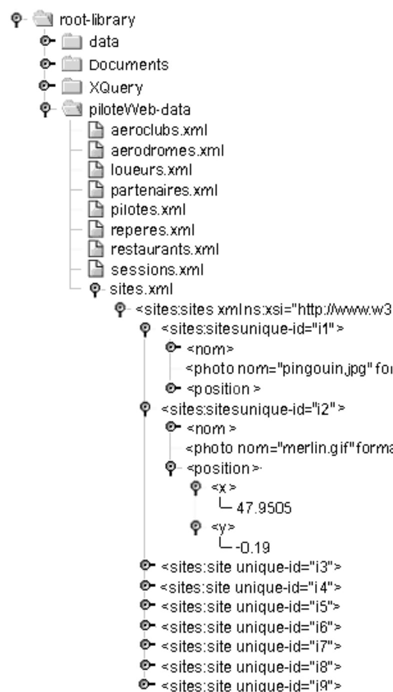


Figure I–4
Arborescence répertoire-fichier-donnée dans une base de données XML

Une conséquence importante en est que les documents XML peuvent être stockés indifféremment dans un système de fichiers et une base de données XML. Les deux formes de stockage sont isomorphes. Contrairement au stockage classique des données, XML permet d'éviter toute forme de transformation des documents XML au moment de leur stockage et de leur dé-stockage.

Cela nous amène naturellement au dernier point de cette section, dans lequel nous allons aborder les qualités de XML dans le domaine de l'archivage.

L'archivage

XML est issu des travaux sur SGML dont les postulats de base étaient, rappelons-le, les suivants : possibilité de lire les fichiers SGML indifféremment par l'homme et la machine, par n'importe quel ordinateur, n'importe quel système d'exploitation, n'importe quelle application.

XML a hérité des mêmes qualités et il en résulte que ce format de données est fiable dans le temps. Il n'y a aucun risque, autre que celui de la dégradation matérielle, qu'un fichier XML créé aujourd'hui ne puisse pas être relu dans cinquante ans. Un document XML n'est lié à aucune application particulière, à aucune version de système d'exploitation particulière, etc.

Avec les bases de données XML (voir figure I-4), les opérations d'archivage bénéficient de la grande facilité avec laquelle on peut choisir la partie de l'arborescence à archiver. Les documents XML à archiver sont extraits de la base de données et deviennent de simples fichiers comme le montre la figure I-5 (l'arborescence exportée est la même que celle présentée à la figure I-4).

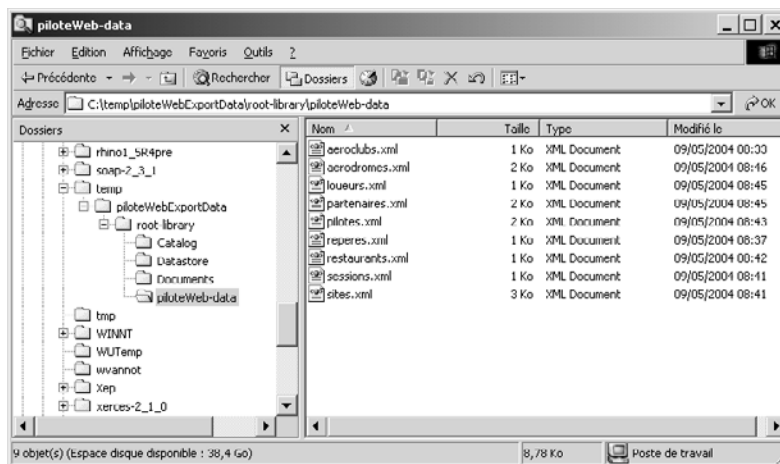


Figure I–5
Export d'une partie d'une base XML vers un système de fichiers

Les fichiers XML ainsi exportés sont bien formés, valides, et contiennent toute l'information nécessaire à leur exploitation présente ou future : la référence au modèle, le type de codage des caractères, les éléments, leurs attributs et tous les objets autorisés dans les documents XML sont présents.

L'application d'archivage peut facilement lire les documents s'agissant de la recherche d'information (de classification par exemple) et de la détermination des entités graphiques qu'il faudra potentiellement archiver avec le document.

Cette facilité d'export des données a des conséquences jusque dans la conception des applications : ce qui est aisé pour un archivage l'est tout autant pour un simple échange de données. Ainsi, ces facilités d'import/export peuvent être mises à profit pour alimenter, à la demande, les différents systèmes constitutifs des infrastructures informatiques, en particulier celles qui sont réparties.

L'expression

Citons un dernier atout de XML, qui a des conséquences non négligeables sur la conception des systèmes d'information : la capacité d'expression naturelle du balisage XML.

Non seulement les données XML sont structurées les unes par rapport aux autres par des relations de type parent-enfant exprimant *de facto* une hiérarchie, mais encore elles sont étiquetées plusieurs fois et de plusieurs manières :

- par le nom des éléments,
- par le nom des attributs,
- par les valeurs des attributs,
- par la hiérarchie des noms des éléments parents,
- par le type associé aux éléments parents,
- par le jeu des identificateurs.

On dispose donc d'une réelle diversité de choix pour spécifier les typologies de reconnaissance des données même s'il revient aux concepteurs d'applications de faire les bons choix.

L'approche qui prévalait ces dernières années, dite par objets métier, n'exploitait qu'une seule des possibilités énumérées plus haut : le nom des éléments. Elle était notoirement insuffisante car :

- Elle créait une relation rigide et bi-univoque entre une classe Java et une balise XML.
- Elle ne permettait de transmettre que des types simples au sens de XML Schema du W3C : le contenu des éléments ne pouvait être qu'une chaîne de caractères (pas de transmission de structures complexes).
- Elle ne permettait pas de gérer le contexte d'utilisation des éléments.
- Elle ne permettait pas de gérer proprement les changements de modélisation et les révisions des applications.

> **VOCABULAIRE Objet métier**
>
> Le principe des objets métier est d'associer automatiquement un nom d'élément à celui d'une classe Java. Ce faisant, un programme recevant une donnée exécutait le programme associé sur la foi du nom de l'élément contenant la donnée.

Insistons sur le fait que ces limitations sont le fruit d'une insuffisance de conception et ne sont en rien dues à XML, qui bénéficie, quant à lui, de possibilités bien plus grandes.

Notamment, les noms des éléments XML sont porteurs d'une sémantique qui peut être polymorphe : puisque la hiérarchie des noms d'éléments forme finalement une suite de mots, les noms donnés aux éléments peuvent changer de signification en fonction de l'endroit où ils se trouvent dans cette hiérarchie ; un mot pris isolément de son contexte n'a pas le même sens qu'un mot « mis en musique ». Il en est de même avec XML.

Par exemple, considérons l'élément `<li>`, représentant généralement un item de liste. Dans la syntaxe xHTML, les séquences `<ol><li>`, `<ul><li>` et `<ol><ul><li>` donnent chacune des interprétations différentes de l'item de liste.

Ce qui est évident pour une caractéristique typographique ne l'est pas moins pour un élément de données. Qu'il s'agisse de reconnaître la règle typographique à appliquer ou le calcul à exécuter, le problème est finalement le même.

Dans l'exemple qui suit, l'élément `date` change de type en fonction de son contexte d'utilisation. Il contient tantôt des sous-éléments (cas ❶) tantôt un simple texte (cas ❷).

```
<period>
 <calculation>
  <date> ❶
   <adjustments>
    <businessDayConvention>MODFOLLOWING</businessDayConvention>
    <businessCentersReference href="#primaryBusinessCenters "/>
   </adjustments>
  </date>
 </calculation>
 <Regular>
 <first>
  <date>2000-10-05</date>❷
 </first>
 <last>
  <date>2004-10-05</date>
 </last>
 </Regular>
 <Frequency base="Interval">
 <Multiplier>6</Multiplier>
 <period>M</period>
 <rollConvention>5</rollConvention>
 </Frequency>
</period>
```

Un programme ne devra évidemment pas traiter l'élément date de la même manière selon qu'il se trouve dans le premier ou le second cas de figure.

Pour conserver cette indépendance entre données et traitements, les noms des éléments doivent être différents de ceux des programmes qui les traitent. Logique et simplicité s'imposent dans la manière de nommer les éléments, ne serait-ce que pour permettre à autrui de lire simplement DTD et instances… L'exemple suivant est un cas d'école de ce qu'il faut éviter de faire :

```
<!ELEMENT N34 - -
(OFFREF?,HOLREF?,DOCID,CDOCID?,H0110,H0151,H0180?,H0170?,H0730+,
     H0860,H0870,H0880?,H0740*,H9750?,H0720?,H0280,H0540,H0510+,H0270,
     H0570?,H0820+,H0300*,H0230*,H0450?,H0152) >

<!--<Title>Industrial design under the 1960 Act-->
<!ELEMENT N60 - -
(OFFREF?,HOLREF?,DOCID,CDOCID?,H0110,H0151,H0180?,H0170?,H0730+,
     H0860,H0870,H0880,H0740*,H9750?,H0720?,H0280,H0540,H0510+,H0570?,
     H0810+,H0820*,H0300*,H0230*,H0450?,GRAPHIC*,H0152) >

<!--<Title>1960 Deposit (monolingual - pre 19959-->
<!ELEMENT N600 - -
(OFFREF?,HOLREF?,DOCID,CDOCID?,H0110,H0151,H0180?,H0170?,H0730+,
     H0860,H0870,H0880,H0740*,H9750?,H0720?,H0280,
     (H054E | H054F),H0510+,(H057E | H057F)?,H0810+,H0820*,H0300*,
     H0230*,H0450?,GRAPHIC*,H0152) >
```

Il revient au concepteur de choisir les règles de nommage adaptées à son cas. Si XML offre un très grand niveau d'adaptabilité, certains modèles peuvent néanmoins coûter très cher en entretien et adaptation.

La pérennité et la flexibilité

Pérennité et flexibilité sont deux qualités qui correspondent respectivement à XML et aux schémas :

- La pérennité indique la capacité des données à durer dans le temps. Elle est garantie par XML.
- La flexibilité représente, de son côté, la capacité des données à s'adapter à des situations imprévues initialement. Elle dépend des schémas.

Des données plus pérennes

Les données et modèles XML sont pérennes *de facto* pour les raisons suivantes :

- Les fichiers XML ne sont pas des fichiers binaires : ils sont lisibles sans décodage préalable par l'ordinateur et un œil humain. Il n'y a donc aucun risque d'erreur ou d'interprétation lors de l'affichage pour lecture humaine ou l'impression d'un fichier XML.
- Tous les ordinateurs du monde respectent et comprennent la codification Unicode des caractères.
- Il n'y a aucun risque d'usure des fichiers avec le temps, un fichier XML ne dépendant strictement d'aucune application. Le seul risque de perte de données avec le temps porte sur la partie mécanique : destruction physique de l'ordinateur ou des médias de stockage. Un fichier XML sauvegardé aujourd'hui restera lisible et exploitable aussi longtemps que ce fichier sera stocké sur un support exploitable.
- Dans le cas le plus extrême, un fichier XML permet de stocker à la fois les données et les explications sur les données : on peut mêler données et documentation relative sans aucune difficulté.

Pour toutes ces raisons, le XML s'impose (et a toujours été recommandé comme tel par les spécialistes) comme format de stockage et d'archivage.

Des modèles plus flexibles

La flexibilité des systèmes est dû à la qualité entourant la mise au point des modèles de données. Les modèles peuvent notamment contenir des articulations qui sont des structures XML permettant, à la manière du squelette humain, de donner de la souplesse aux modèles.

En fonction des traitements à effectuer, les documents XML peuvent être découpés aux articulations. Les documents XML s'adaptent ainsi à un plus grand nombre de cas d'applications ; la comparaison avec le jeu de Lego™ est évidente, mais significative. Notamment, cela permet aux documents XML de s'adapter à d'autres modèles de données et processus de traitement. La flexibilité ne se limite pas au stockage statique des données, mais concerne toute la dynamique du système d'information.

En résumé

Dans ce chapitre introductif, les thèmes autour desquels s'articule notre ouvrage ont été abordés.

Nous avons tout d'abord introduit la notion de modèle. Invitant à réfléchir sur le sens du mot modèle, nous avons rappelé qu'il peut désigner une représentation synthétique de la réalité ou un moule visant à produire une copie exacte de l'original.

Dans le cadre de la représentation du système d'information, il est apparu nécessaire de disposer d'un langage de modélisation de haut niveau pour encadrer l'emploi de XML. UML est apparu comme le langage de référence pour la représentation conceptuelle des données.

Nous avons ensuite rappelé les apports de XML à la modélisation, qui ont deux origines : l'une provenant du modèle XML lui-même, l'autre du travail de normalisation.

D'une manière plus générale, XML dispose de qualités intrinsèques qui font de lui un excellent candidat pour les systèmes d'information. Au travers de différentes sections, sa facilité d'adaptation, ses capacités d'adressage, de stockage, d'archivage, d'expression ont été détaillées. Et nous avons également fondé notre discours sur la pérennité, la flexibilité des modèles, et l'efficacité avec laquelle XML peut être mis en œuvre.

Nous nous sommes attachés à montrer ce qui fait la puissance et l'omniprésence de XML dans les systèmes d'information. Au-delà de la seule curiosité technologique, nous avons présenté les arguments objectifs justifiant l'organisation des futurs systèmes autour de la technologie XML.

À partir du chapitre suivant, nous allons rentrer dans le vif du sujet, en l'initiant par une démarche d'analyse du projet.

Première partie

Étapes de la démarche de modélisation

1

Étape 1 - La préparation du projet

Comme nous l'avons souligné en introduction, XML permet d'augmenter la flexibilité des systèmes d'information. Toutefois, un tel résultat ne s'obtient pas sans plan de travail : une méthodologie adaptée est nécessaire. Commençons donc par le début : la réalisation d'une esquisse du futur système.

Ce chapitre est consacré à ce que vous devez faire dans les tout premiers temps de votre projet. On pourrait appeler cette période le projet de projet.

Nous y expliquons comment réaliser rapidement un premier dégrossissage du problème qui se présente à vous. Cela doit vous permettre non seulement de comprendre le projet, mais encore de le partager avec d'autres alors que seules des idées sont formulées (à ce stade, rien de concret n'existe encore vraiment).

Nous présentons ici la méthode qui permet d'obtenir rapidement l'esquisse du projet en tenant compte, dès le départ, de grands principes que nous expliquons au cours de ce chapitre.

Inspirés des méthodes de conception d'*architectures orientées services*, les croquis (ou esquisses) que vous obtiendrez représenteront *in fine* les fonctions, sous-fonctions et services qui composeront le futur système.

Bref aperçu de la méthode

Objectifs

L'objectif est de préparer le projet : rationaliser le travail dès le début car cela se traduira finalement par un gain de temps.

De plus, fournir des esquisses (graphiques) est efficace pour présenter le projet :

- À sa hiérarchie : rationalisé, le projet n'en sera que plus convaincant.
- Aux utilisateurs : une représentation simple et compréhensible des fonctions améliorera la qualité des discussions.
- À l'équipe de développement : le rôle de chacun et les expertises attendues seront bien identifiés.

Ainsi, ce travail de préparation du projet va vous apporter :

- de l'aisance quand vous aurez à parler de votre projet ;
- de l'assurance car vous pourrez comparer différentes solutions ;
- de l'efficacité en avançant plus rapidement qu'en utilisant les méthodes classiques ;
- la possibilité de contrôler la complétude du système (n'oublier aucune fonction) ;
- la capacité à évaluer la quantité de travail de développement à réaliser ;
- un support utile pour organiser les développements ;
- une bonne maîtrise des différences entre les aspects fonctionnels et techniques du futur système.

On le voit ici, l'objectif recherché est celui de la clarification d'un problème.

Puisqu'elle permet d'étudier le système plus ou moins en profondeur (effet de zoom), la méthode laisse la liberté de le représenter de manière plus ou moins macroscopique. C'est à vous de choisir, mais n'oubliez toutefois jamais le principe des cartes routières : schématiser la réalité sans la représenter totalement ! (si une carte routière était égale à la réalité qu'elle représente, elle serait aussi grande que cette réalité…). Le but d'une telle méthode est d'obtenir un schéma compréhensible du futur système.

La phase de préparation du projet, objet de ce chapitre, vise à vous fournir le plus rapidement possible une vue schématique des fonctions et services de votre futur système.

Principes généraux

De la donnée élémentaire stockée dans une base de données à la page HTML, d'un plan de bureau d'étude à la description d'une opération de montage/démontage, des données de spécification d'emballage aux documents décrivant les conditions de transport, les exemples ne manquent pas du long et permanent processus de traitement de

l'information. Maintes fois répété, il est toujours le même : des informations brutes sont choisies pour former un ensemble informatif cohérent qui, à son tour, est transmis sous la forme de documents (des informations rendues lisibles) à des personnes (ou systèmes informatiques) capables de les interpréter et de les utiliser à bon escient.

À partir de ce constat, la méthode repose sur quatre principes élémentaires :

- Le processus d'extraction, assemblage et publication est invariant et caractérise tout système d'information.
- Au sein d'un système d'information, le composant élémentaire est un service capable de recevoir une information, la transformer et la transmettre à son tour à un autre composant élémentaire. Tout composant élémentaire est constitué d'au moins deux fonctions : l'une sert à transmettre, l'autre à effectuer un traitement.
- Information et rigidité sont deux mots antinomiques. Le maître mot est *flexibilité*.
- Pour faire face à la complexité du problème, il est impératif de pouvoir travailler sur des représentations simples : synthétiser pour mieux comprendre.

De manière plus concrète, la méthode met en évidence les fonctions du système et, *via* un mécanisme de classification et d'attribution de points, transforme une liste de fonctions en courbes et histogrammes explicites.

De par l'importance qu'elle donne aux échanges d'information, la méthode se distingue de celles fondées sur UML et l'analyse des cas d'utilisation (*use cases*) du système. Ces dernières cernent les objets à partir des différents modes d'utilisation du système étudié, chacun étant décrit à l'aide d'*un cas d'utilisation* d'UML. C'est alors en s'appuyant sur ces cas d'utilisation que le concepteur fera émerger très tôt les principes de la conception orientée objet : l'encapsulation des données et des traitements par des classes. L'accent mis dès le début sur les classes contribue alors implicitement à privilégier l'analyse des traitements plutôt que celle des données échangées. La primauté donnée au composant objet conduit ainsi à un typage des données orienté traitement qui rend plus difficile une conception orientée vers l'autonomie des données échangées, garante de plus de flexibilité et de pérennité. L'échec du déploiement à grande échelle des architectures d'objets distribuées a contribué au développement des architectures basées sur les services où la coordination entre les différentes unités de traitement, appelées « services », est réalisée à l'aide d'échanges d'information. C'est ce principe d'analyse des échanges qui sert de base à la démarche proposée dans ce chapitre.

Limites

« *L'utilité de la méthode s'arrête là où commence celle des autres* », à savoir :

- Elle n'a pas la prétention d'être une théorie générale de la modélisation des systèmes d'information. Sa seule vocation, et avantage, est d'être simple, pratique et

rapide à mettre en œuvre. Elle est limitée au travail de réalisation d'une esquisse du futur système.

- Elle ne concerne que la première des étapes permettant d'aboutir à la modélisation complète du système. D'autres suivront d'où sortiront, par exemple, des schémas XML, des modèles de traitement et de stockage. À ce titre, elle ne couvre pas les considérations relatives à la mise en œuvre physique du futur système.

Contexte idéal d'utilisation de la méthode

Cette section présente quelques définitions permettant de mieux cerner le contexte technique et conceptuel dans lequel les résultats obtenus avec la méthode sont les meilleurs.

Un *système de traitement de l'information* est un ensemble constitué d'une ou plusieurs application(s) informatique(s), capable de transformer en information des données élémentaires. Délivrer cette information à des utilisateurs est sa principale raison d'être, quelle que soit la forme physique choisie pour remettre cette information.

Le terme *information* est volontairement utilisé plutôt que *donnée.* Nous entendons par information un ensemble de données formant une *unité logique d'information*.

Une *unité logique d'information* est un ensemble de données auxquelles on a donné du sens en suivant une logique informative. L'unité logique d'information ressemble à un document, à savoir une entité ayant un sens informationnel pour l'humain qui la lit dans un contexte particulier. Cependant, cela peut également être une partie d'un document : par exemple, l'ensemble des données composant une commande, abstraction faite de celles concernant le destinataire de cette commande (et vice versa).

Les *données* peuvent être de type *unitaire*, *documentaire* ou *graphique*, c'est-à-dire qu'elles peuvent correspondre à des champs d'une base de données, des paragraphes textuels et des objets visuels. Isolée, une donnée n'a qu'un sens tandis qu'associée à d'autres, elle devient *information* et peut avoir plusieurs sens.

Présentation détaillée des quatre principes de base de la méthode

Principe n°1. Un système d'information repose toujours sur un même modèle de base.

Le premier principe de base est construit autour de l'idée que l'information est une matière vivante. À ce titre, elle a un début, une vie et une fin : elle naît, grandit et meurt.

Ce cycle de vie est simple à représenter. En termes techniques, le début d'une information se situe dans un domaine baptisé « production » tandis que sa croissance se passe dans un « vivier » que nous avons baptisé « gestion ». Quant à sa fin, nous la faisons commencer dès que l'information est diffusée ; en effet, elle échappe alors au système qui l'a vu naître et passe dans un autre monde.

C'est pourquoi la représentation la plus élémentaire de notre système d'information est constituée de trois zones, représentées à la figure 1-1.

Figure 1–1
La représentation la plus élémentaire de notre futur système d'information

C'est à partir de ce socle que nous allons pouvoir construire une première représentation de notre système, brique par brique en suivant la démarche suivante :

- identifier les fonctions du système,
- placer chaque fonction au dessus du socle qui lui convient,
- empiler les fonctions selon un plan (que nous allons vous présenter plus loin),
- construire l'ensemble en pensant que chaque fonction doit rendre un service potentiellement indépendant de tous les autres.

C'est ce dernier point qui est détaillé dans le principe n°2.

Principe n°2. Un système d'information est un ensemble de fonctions et services.

> **VOCABULAIRE Service**
>
> Selon les principes des architectures orientées service, un service est une unité de traitement autonome communiquant avec son environnement à l'aide de messages.
> Les services définissent ainsi des frontières entre les lieux de traitement de l'information, l'intérieur des services, et les lieux d'échanges de l'information – la communication entre les services.

> **VOCABULAIRE Fonction**
>
> On peut définir une fonction comme étant un traitement permettant de produire un résultat spécifié à partir d'une entrée également spécifiée.

Les définitions des mots « service » et « fonction » sont très proches l'une de l'autre. À partir de la définition des services, on passe très facilement à celle de fonction. En com-

munication, la différence entre les deux est infime. En plaçant la question de la communication de l'information au cœur de la problématique de conception (architecture) des systèmes, on fait ressortir le rôle des services, à savoir les fonctions du système qui vont ingérer, transformer et traiter l'information reçue sous forme de documents XML. En ce sens, les architectures orientées services et l'échange d'information (de documents) au format XML sont deux composantes non seulement complémentaires mais consubstantielles des systèmes d'information objets de cet ouvrage.

Vous allez voir que, grâce à ce principe, on peut obtenir une bonne représentation des échanges d'information au sein du futur système. La mise en valeur des communications entre les services appartenant aux différentes zones du socle illustré à la figure 1-1 – production, gestion, diffusion – met en évidence les étapes les plus significatives des différentes phases de traitement et d'enrichissement de l'information. Nous verrons ultérieurement qu'il existe une relation de cause à effet directe entre la qualité avec laquelle ces communications inter-zones sont identifiées et la flexibilité finale du système. Cela implique une relation de cause à effet directe ente la qualité de cette conception et le coût de développement et maintenance du système final.

Dans la figure 1-2, le système est représenté comme s'il s'agissait d'une usine de traitement de l'information : à gauche y entrent des granules d'information, des données, et à droite sort un produit raffiné qui est bien souvent un document.

Au dessus de la brique « production » du socle vont se retrouver toutes les fonctions qui servent à créer ou récupérer les granules d'information initiaux.

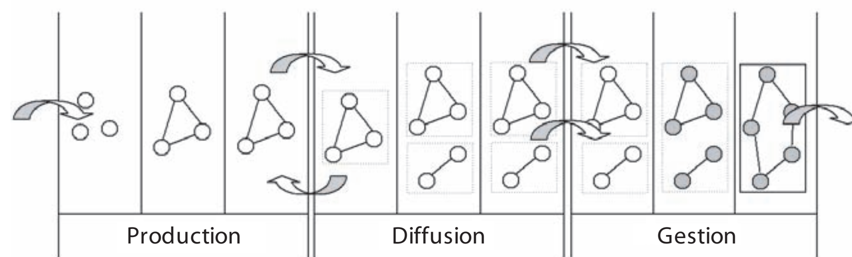


Figure 1–2
Fonctions et services composent un système d'information.

Au dessus de la brique « gestion » seront représentées toutes les fonctions d'assemblage, de gestion de la configuration et des processus de fabrication d'une information finie.

Au dessus de la brique « diffusion » seront empilées toutes les fonctions permettant de diffuser l'information. Dans la figure 1-2, nous essayons de montrer la logique avec laquelle nous allons construire une représentation efficace de notre futur système, mêlant petit à petit les notions de fonctions et de services. De granules blancs et isolés les uns des autres à l'entrée du système, on montre comment ils sont assemblés, placés par rapport à un contexte extérieur dans la brique « gestion » (contexte

symbolisé par un rectangle en pointillé), puis comment ils sont adaptés à un contexte de diffusion (les granules blancs deviennent gris).

Le tout respecte le troisième principe, celui de la flexibilité, décrit dans la prochaine section.

Principe n°3. Un système d'information est obligatoirement flexible.

L'information, presque par définition, est en évolution permanente. Au cours du temps cependant, les traitements changent à un rythme encore plus rapide que les données. Ces dernières doivent donc être organisées de manière à assurer au système une grande flexibilité dans le temps.

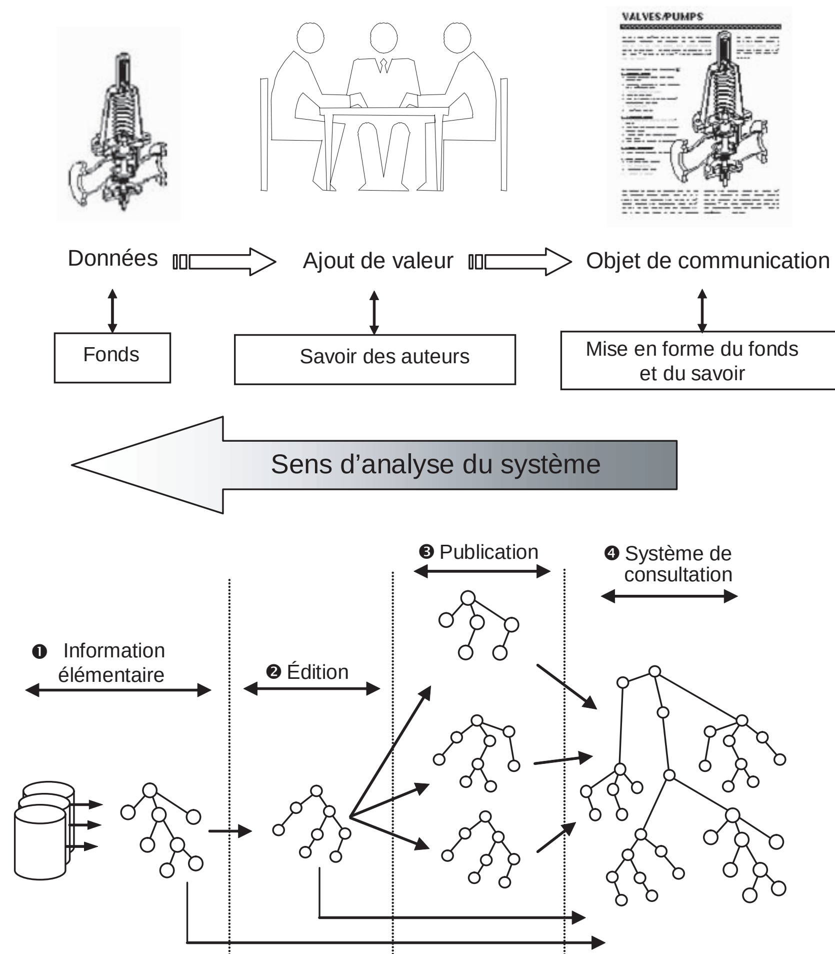


Figure 1–3
L'information DOIT être articulée et transportable.

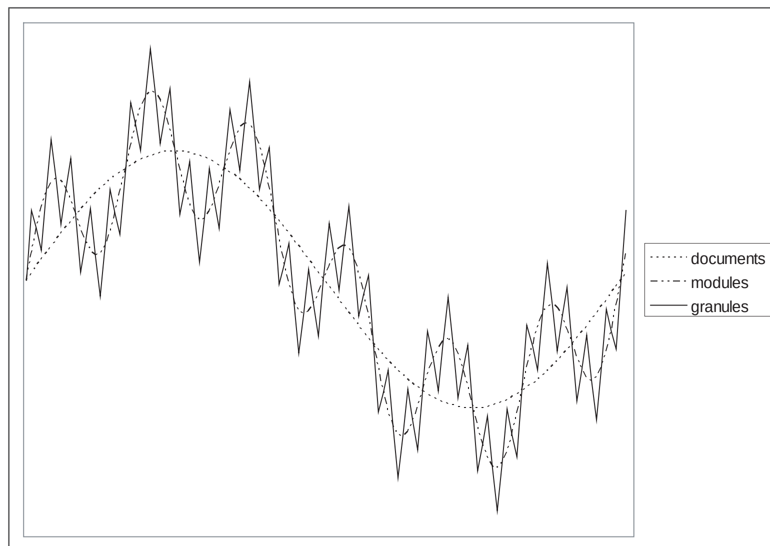
Si vous respectez le principe n°2, vous aurez à l'esprit que votre système manipulera des granules éminemment transportables alors que dans les méthodes de conception classiques, on a plutôt tendance à figer l'information dans des cases…

La flexibilité d'un système d'information équivaut aux articulations du squelette humain : elle confère au système, non des possibilités de mouvement, mais d'adaptation de l'information. Elle permet d'utiliser l'information à plusieurs fins, sur plusieurs médias, avec différents traitements de transformation. La figure 1-3 schématise plusieurs idées relatives à la flexibilité :

- Le système doit s'adapter (en permanence) à l'évolution des organisations humaines.
- Le système doit s'adapter aux besoins des consommateurs de l'information et de l'évolution du marché qu'ils représentent.
- Le système doit permettre de réutiliser une même information, ou granule d'information, ou donnée élémentaire.

Enfin, la figure 1-4 schématise un autre type de flexibilité : celle du rythme des modifications des informations gérées par le système. Certaines informations ou données seront modifiées très régulièrement (des cours de bourse par exemple), alors que les documents qui contiendront ces données seront potentiellement mis à jour selon un rythme différent. Toujours en prenant l'exemple traditionnel des données boursières, pensons aux documents papier contenant des analyses financières ou les pages Web qui affichent les cours : on accepte la mise à jour régulière de la page, mais on n'accepte pas que sa forme change continuellement.

Figure 1–4
Le système doit permettre de prendre en compte les fréquences de mise à jour propres à chaque type d'information.



Principe n°4. La représentation du système d'information doit être synthétique et simple.

La représentation graphique du système repose sur des règles simples. Cela facilite la comparaison de différentes solutions. Cette simplicité est donc un but recherché. Avec une représentation trop complète ou trop sophistiquée, cette comparaison serait impossible : le seul temps passé à établir de telles représentation ferait perdre l'intérêt même de la comparaison. Un exemple typique est celui des tarifs des opérateurs de téléphonie : à vouloir être trop exhaustifs dans leurs tarifs, toute comparaison globale devient impossible ! C'est ce que nous voulons éviter avec notre méthode.

D'après la méthode, le système va être progressivement découpé en services (ensembles de fonctions de traitement) et échanges de données entre services. Un système simple d'attribution de points permet de noter chaque fonction et chaque service. La grille d'attribution des points ainsi obtenue permet d'associer un coût théorique à chaque solution étudiée : plus le nombre de points est élevé et plus la solution sera coûteuse à réaliser. La valeur des points aide ainsi à se faire une idée du coût de réalisation du projet.

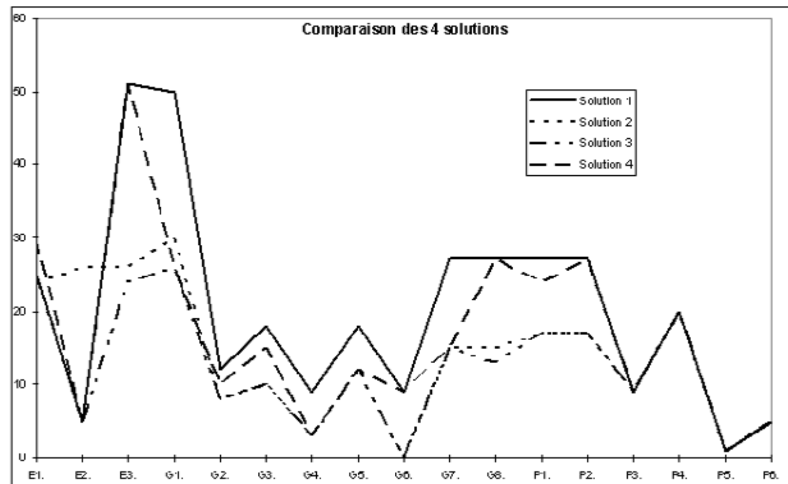


Figure 1–5
Courbes représentant le système au terme de la phase d'ébauche

Au terme de sa mise en œuvre, la méthode produit des courbes parlantes, représentant le système ébauché. Par exemple, la figure 1-5 fournit le graphique obtenu au terme de la pré-étude d'un système qui devait assurer la gestion de configuration entre des systèmes mécaniques ayant de multiples options et la documentation décrivant justement lesdites options. Pour cette étude, quatre hypothèses technologiques différentes avaient été retenues. Les courbes obtenues avec la méthode proposée dans ce chapitre ont permis de comparer en un clin d'œil le coût respectif de chacune de

ces hypothèses. Plus précisément, la représentation a permis de comprendre et discuter chaque poste de coût, fonction par fonction, service par service.

Mise en œuvre de la méthode

La mise en œuvre de notre méthode se déroule en trois ou quatre phases selon la profondeur d'analyse recherchée (figure 1-6). Dans le cas de systèmes simples, l'étape d'analyse des services peut être omise. Dans les systèmes plus complexes, l'identification des services, c'est-à-dire des canaux de communication, donne le moyen de maîtriser les flux d'information. Cette étape devient carrément indispensable dans le cas des systèmes complexes ayant de nombreuses interactions avec d'autres systèmes.

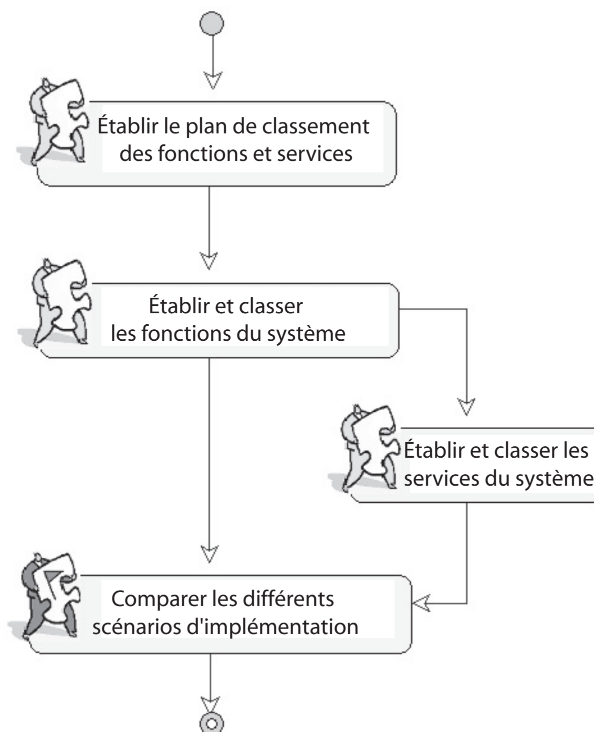


Figure 1–6
Représentation des différentes étapes de la mise en œuvre de la méthode

Au stade ultime de la méthode, on peut établir des tableaux comparatifs de différentes solutions techniques. En fin de chapitre, la présentation d'un cas réel en donnera un exemple.

Étape n°1. Établir un plan de classification des fonctions et services du système

Comme nous l'avons abordé dans les premières sections de ce chapitre, la définition la plus élémentaire d'un système d'information est constituée des trois zones production, gestion et diffusion. Invariablement, le cycle de vie de toute donnée, fichier ou document s'inscrira dans les fonctions et services qui s'y trouvent définis.

La zone de production regroupe les fonctions associées à la naissance de l'information. Cela peut être le résultat d'une importation depuis un autre système, d'une conversion, d'une génération spontanée consécutive à un calcul, ou encore d'une action manuelle humaine comme la saisie d'une donnée ou d'un document.

La zone de gestion regroupe les fonctions associées au stockage de l'information dans le système où elle sera contrôlée, protégée, mise dans un processus et archivée. Cette zone sert à garantir la qualité et la pertinence de l'information. Pour cela, une fonction importante de la zone est la gestion de configuration : le système doit permettre de garantir la cohérence des informations les unes par rapport aux autres.

La zone de diffusion regroupe les fonctions qui servent à préparer l'information aux médias de diffusion, en faire des pages HTML ou papier ou transmettre l'information à un autre système.

Chaque zone est composée d'un certain nombre de fonctions, regroupées en blocs. La méthode propose une série de blocs standards, ce qui ne vous empêche pas ultérieurement d'ajouter les vôtres. Ceux que nous proposons sont très utiles lors du démarrage de l'étude de conception d'un système.

La figure 1-7 présente les blocs standards de la méthode, ceux que l'on retrouve immanquablement dans tout système d'information

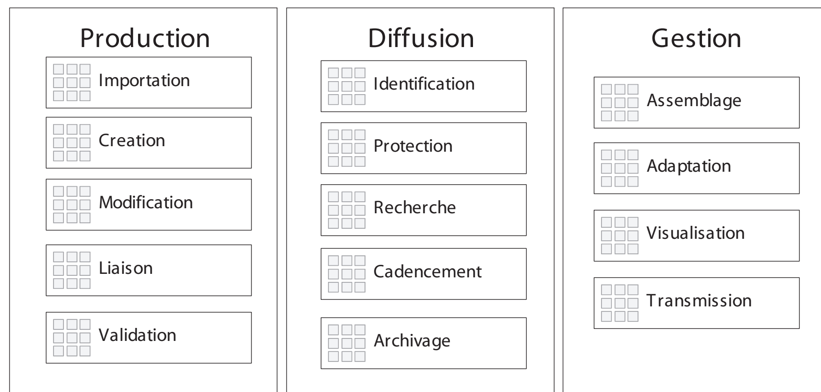


Figure 1–7
Les blocs fonctionnels standards de la méthode

Pour la zone *production*, les blocs standards sont les suivants :

- *Importation* : fonctions de récupération des données d'un autre système, logiciel ou base de données.
- *Création* : fonctions de création des objets de type donnée, texte, image, illustration etc.
- *Modification* : fonctions d'édition des objets créés dans les blocs *création* et/ou *importation*.
- *Liaison* : fonctions de pose de liens entre les objets.
- *Validation* : fonctions de contrôle de conformité des objets déposés dans le système d'information.

Pour la zone *gestion*, les blocs standards sont les suivants :

- *Identification* : fonctions d'étiquetage des objets stockés dans le système. Ces données d'identification servent la plupart du temps à retrouver, classer et gérer les objets contenus dans le système. Dans le monde des documents, les données d'identification sont souvent appelées *méta-données*.
- *Protection* : fonctions de contrôle d'accès aux objets par la définition de droits. Des programmes, autant que des humains, sont susceptibles d'exploiter les objets se trouvant dans le système. Les droits d'accès peuvent changer en fonction d'événements divers : une même personne, ou un programme, peut voir ses droits évoluer en fonction de la propre évolution de l'objet auquel accéder.
- *Cadencement* : fonctions de définition et d'activation des processus composant le cycle de vie des objets, tous susceptibles d'être soumis à des circuits de validation particuliers avant d'être publiés.
- *Recherche* : fonctions de recherche et d'analyse des objets de la base pour en garantir l'intégrité ou tout simplement les retrouver.
- *Archivage* : fonction de recopie (voire destruction) des objets du système pour en faire des copies de sécurité ou les rendre hors d'usage tout en en conservant la trace.

Pour la zone *diffusion*, les blocs standards sont les suivants :

- *Assemblage* : fonctions d'agrégation et collecte des objets afin d'en faire un tout cohérent pour un média de restitution et un objectif d'information.
- *Adaptation* : fonction de transformation des objets pour les adapter non seulement aux médias de diffusion mais encore aux lecteurs ou systèmes informatiques destinataires de l'information.
- *Visualisation* : fonctions d'affichage des documents obtenus par les fonctions d'assemblage et adaptation.
- *Transmission* : fonction de transport des documents obtenus jusqu'à leurs destinations finales.

C'est en adaptant ce plan générique « d'urbanisation » au cas de votre propre système d'information que vous allez peu à peu le structurer et trouver ses spécificités.

Étape n°2. Identifier et classer les fonctions du système

Dans les projets informatiques, il est classique d'analyser les fonctions et sous-fonctions d'un système. Cette étape repose tant sur les demandes des utilisateurs que sur le propre savoir du concepteur du système. En effet, si les utilisateurs connaissent leurs besoins, ils ne sont que rarement capables de les mettre en regard d'une logique informatique. Il revient au concepteur de faire ce travail d'appairage.

Pour conduire cette étape, il faut partir des demandes fonctionnelles des utilisateurs. On établit alors une liste de fonctions et sous-fonctions, si possible en limitant à 1 le nombre de sous-niveaux de fonctions (interdisez-vous d'avoir des sous-sous-fonctions).

Une fois cette liste établie (le tableau 1-1 en donne un exemple), chaque fonction ou sous fonction est associée à l'une (et une seule) des trois zones élémentaires du système. Dans la colonne *zone*, on met un P quand la fonction correspond à la zone *production*, un G quand il s'agit de la zone *gestion* et un D pour la zone *diffusion*.

Dans le tableau, vous observerez également que chaque fonction ou sous-fonction se voit attribuer un code d'identification (FP13 par exemple pour la fonction « créer une structure de document en XML »).

Le tableau 1-1 fournit un exemple du résultat à obtenir pour trois des fonctions principales d'un système. Il ne s'agit que d'un extrait, un système réel étant composé de plusieurs dizaines de fonctions principales.

À la manière de XML Schema, qui ne fournit pas un schéma universel mais est un outil pour créer tout type de modèle, notre méthode ne fournit pas une décomposition exhaustive d'un système. C'est en ce sens qu'elle n'a pas la prétention d'être une théorie générale de l'architecture d'un système d'information. Notre méthode est un outil pour construire votre propre représentation de votre système : à vous d'identifier les fonctions principales et secondaires qui caractériseront votre système selon le plan générique que nous fournissons et qui, quant à lui, est invariant.

Tableau 1–1 Exemple de fonctions identifiées lors d'une étude préalable de besoin

Réf. interne		Identification des fonctions	Zone
FP1		Manipuler des structures de données et produire les fichiers XML correspondants.	
	FP11	Importer dans la base une structure de données définie extérieurement.	P
	FP12	Créer des squelettes de publication.	P
	FP13	Créer une structure de document en XML.	P

Tableau 1–1 Exemple de fonctions identifiées lors d'une étude préalable de besoin (suite)

Réf. interne	Identification des fonctions	Zone
FP14	Comparer (rechercher et analyser) deux structures de données.	G
FP15	Connaître les impacts de l'évolution d'une structure (rechercher et analyser).	G
FP16	Créer des structures de données de test.	P
FP17	Modifier des squelettes de publication.	P
FP18	Valider les unités d'information.	P
FP2	Gérer des structures de publication.	
FP21	Créer des modèles réutilisables de structures de publication.	P
FP22	Rechercher des structures de publication.	G
FP23	Contrôler les accès aux différentes fonctions relatives à la gestion des structures de publication.	G
FP3	Manipuler des unités d'information et des graphiques.	
FP31	Disposer d'un programme permettant d'introduire des illustrations dans la base.	P
FP32	Valider les unités d'information entrant dans la base. Pour les unités d'information XML, il s'agit d'utiliser un parseur. En cas d'erreur, un opérateur pourra intervenir.	P
FP33	Convertir en XML les unités d'information non-XML. Convertir les illustrations aux standards retenus.	P
FP34	Éditer des unités d'information.	P
FP35	Disposer d'un programme permettant d'introduire dans la base des unités d'information.	G
…	…	…

Ce travail étant fait, l'étape suivante consiste à reporter chaque fonction sur un histogramme (figure 1-8) qui va fournir une première représentation visuelle du système. Ce placement est fait selon une logique que nous expliquons dans la suite de cette section.

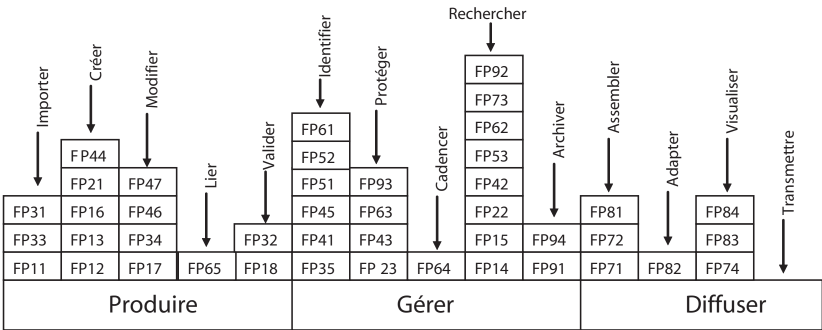


Figure 1–8
Représentation sous forme d'histogramme des fonctions du système

Les fonctions sont placées en respectant la logique suivante :

- Le socle de base de l'histogramme représente les zones.
- Au dessus de chaque zone sont posées les fonctions identifiées au tableau 1-1 : une brique par sous-fonction. Chaque colonne correspond à l'un des blocs définis pour chaque zone (*importer*, *créer*, *modifier etc.)*.
- S'il se trouve des fonctions d'interfaçage entre les zones, telles que *production* et *gestion*, elles seront placées de part et d'autre de la ligne verticale qui sépare les zones concernées. Lorsque cela est possible, les fonctions qui se répondent seront placées en regard les unes des autres. Par exemple, les fonctions FP18 (*validation des unités d'information*) et FP35 (*programme d'importation des unités d'information validées*) se correspondent dans notre exemple (en général, c'est le concepteur ayant établi la liste des fonctions qui peut le dire) : elles sont donc placées en regard l'une de l'autre.
- L'ordre vertical n'est pas imposé par la méthode : choisissez votre propre règle si cela permet à votre modèle d'être encore plus explicite. Vous pouvez par exemple les positionner en fonction de leur ordre de dépendance les unes avec les autres, ou encore en mettant celles qui vous semblent les plus importantes en bas et en terminant en haut par d'éventuelles options du système.
- À l'extrême gauche de l'histogramme se trouvent les fonctions d'importation dans le système tandis qu'à l'extrême droite sont placées les fonctions de diffusion vers l'extérieur. Cela permet d'avoir une compréhension rapide du graphe : à gauche sont représentées toutes les tâches nécessaires pour alimenter le système et à droite toutes celles pour émettre l'information. Au centre se trouvent les fonctions de gestion.

Étape n°3. Identifier et classer les services du système

Cette étape concerne les systèmes de bonne taille. Quand le système est simple, l'analyse fonctionnelle peut suffire et cette étape est alors inutile.

L'analyse fonctionnelle traditionnelle ne met que rarement en évidence les interactions ayant lieu tant à l'intérieur du système qu'entre le système et son environnement. En effet, cette question est essentiellement technique et non fonctionnelle. Le but de l'étape 3 est justement d'identifier ces interactions car elles conditionneront l'organisation des composants logiciels, ou services dans ce cas, du système. Les équipes techniques pourront alors développer une architecture *orientée service* du système sans perdre de vue son objectif fonctionnel.

Les services sont des unités autonomes de traitement, coordonnées par des messages. Ils organisent les fonctions découvertes lors de l'étape précédente pour faire apparaître les canaux de communication du système. L'organisation des services, c'est-à-

dire la manière dont ils vont intervenir dans la vie du système, représentera la manière dont les fonctions interagissent entre elles, en particulier en ce qui concerne les données ou objets échangés. Les services vont ainsi dévoiler un aspect très important de la conception : le format, ou modèle, des données échangées entre les fonctions du système. Il y a ainsi une étroite relation entre les services et la conception des modèles de données.

En complément de l'analyse purement fonctionnelle, l'identification des services revient à bien isoler tous les points de transformation ou d'échange d'objets (principalement des données ou des informations). Elle complète et organise utilement la liste des fonctions identifiées lors de la phase précédente.

Une attention particulière doit être portée aux services situés à la frontière entre les zones de production, de gestion et de diffusion. Les échanges entre ces services représentent les points d'articulation les plus sensibles du système. Leur bon recensement est une condition requise pour l'élaboration de scénario d'implémentation de qualité.

Étape n°4. Comparer les scénarios d'implémentation

La dernière partie de la méthode va aboutir au chiffrage des différentes solutions d'implémentation. Pour cela, nous vous proposons une démarche qui vous permettra d'attribuer des points aux fonctions et services identifiés lors des étapes précédentes.

Pour que l'analyse du système soit objective, les points seront choisis en fonction de critères objectifs !

L'analyse va consister à donner à chaque fonction et service du système étudié une série de cinq notes qui représentent :

1. le poids fonctionnel (l'acronyme utilisé, FW, est celui de *Function Weight*) ;
2. le poids logiciel (ou SW comme *Software Weight*) ;
3. le poids du développement (ou DW comme *Development Weight*) ;
4. le coût du développement (ou DC comme *Development Cost*) ;
5. le coût du développement avec prise en compte de la qualité (ou DCQ comme *Development Cost with Quality*).

Nous allons voir maintenant comment appliquer les notes.

Le *poids fonctionnel* est une note qui vaut `0` quand la fonction est optionnelle ou inutile, et `1` quand elle est impérative. Il ne s'agit pas uniquement de mesurer par là l'intérêt d'une fonction par rapport au système cible, mais de faire apparaître des différences entre plusieurs solutions techniques. Par exemple, il se peut qu'une fonction de transformation de données soit nécessaire dans le cas d'une solution technique et inutile dans l'autre.

Le *poids logiciel* est une note qui peut prendre les valeurs 0, 1 ou 2. La note 0 est attribuée quand la fonction est optionnelle ou inutile et que son poids fonctionnel est déjà 0. La note 1 est attribuée aux fonctions mises en œuvre, de base, dans la solution technique concernée. La note 2 signifie que la fonction n'est pas de base dans la solution technique envisagée et doit faire l'objet d'un paramétrage ou d'un développement spécifique.

Le *poids du développement* est une note comprise entre 0 et 3 : la note 0 est conservée pour les fonctions optionnelles ou inutiles, la note 1 pour les fonctions de la solution étudiée, la note 2 correspond aux fonctions qui peuvent être mises en œuvre par un simple paramétrage tandis que la note 3 correspond aux fonctions qui nécessitent un développement spécifique.

Le *coût du développement* est l'évaluation du temps nécessaire pour développer les fonctions dont le poids du développement est 3. Pour les autres fonctions, la note correspondant au poids du développement est maintenue.

Enfin, le poids de la qualité est le poids du développement augmenté d'un coefficient qualité multiplicateur à choisir en fonction du niveau de qualité exigé pour l'application et des habitudes de votre société sur ce point.

Les tableaux 1-2 et 1-3, ainsi que la figure 1-9, illustrent les résultats obtenus. Le tableau 1-2 présente la liste comparative des fonctions identifiées pour les besoins d'un système de traitement de l'information pour lequel quatre solutions techniques – baptisées sol1, sol2, sol3 et sol4 – ont été étudiées. Ce tableau est un extrait d'une étude réelle complète portant sur une centaine de fonctions.

Les colonnes de droite du tableau contiennent les 0 et 1 du poids de chaque fonction dans chacune des solutions envisagées.

Tableau 1–2 Liste comparative des fonctions de quatre solutions

Liste des fonctions de chaque solution							
Zone	**MF**	**SF**	**Description**	**Sol1**	**Sol2**	**Sol3**	**Sol4**
PRODUCTION							
	E1	Importer					
		FP11	Importer dans la base une structure de données définie extérieurement.	1	1	1	1
		FP33	Convertir en XML les unités d'information non-XML. Convertir les illustrations aux standards retenus.	1	1	1	1
		FP31	Disposer d'un programme permettant d'introduire dans la base des unités d'information.	1	1	1	1
	E2	Créer					
		FP12	Créer des squelettes de publication.	1	0	1	1
		FP13	Créer une structure de données en XML.	1	0	1	1

Tableau 1–2 Liste comparative des fonctions de quatre solutions (suite)

Liste des fonctions de chaque solution							
Zone	**MF**	**SF**	**Description**	**Sol1**	**Sol2**	**Sol3**	**Sol4**
GESTION							
	G1	Identifier					
		FP35	Disposer d'un programme permettant d'introduire des illustrations dans la base.	1	1	1	1
		FP41	Gérer les unités d'information et les graphiques associés.	1	1	1	1
		FP45	Gérer les chemins d'accès aux unités d'information qui seraient stockées à l'extérieur de la base.	1	1	1	1
	G2	Protéger					
		FP23	Contrôler les accès aux différentes fonctions relatives à la gestion des structures de publication.	1	1	1	1
		FP43	Contrôler les accès aux unités d'information (notion de workflow).	1	1	1	1
		FP63	Contrôler les autorisations d'accès aux fiches d'identité.	0	1	1	1
		FP93	Pouvoir mettre les publications dans une zone protégée du système.	0	0	0	1
	G3	Cadencer					
		FP64	Gérer les processus.	1	1	1	1
DIFFUSION							
	P1	Assembler					
		FP71	Produire une structure de données en une instance XML contenant l'ensemble des textes.	1	1	1	1
		FP72	Pouvoir effectuer cet export selon différents critères de sélection.	0	1	1	1
		FP81	Pouvoir choisir parmi plusieurs formats de présentation.	0	0	1	1
	P2	Adapter					
		FP82	Disposer des convertisseurs XML->papier.	1	1	1	1
	P3	Visualiser					
		FP74	Visualiser des unités d'information ou des graphiques depuis l'interface de gestion de la base.	1	1	1	1
		FP83	Disposer d'un navigateur.	0	1	1	1
		FP84	Prévoir un poste de contrôle des versions papier ou électroniques produites.	1	1	1	1
	P4.	Transmettre					
…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…

TOTAUX	28	31	36	41

Le tableau 1-3 représente toutes les notes obtenues par l'une des solutions étudiées (à savoir la solution 3). L'on y retrouve, pour chaque fonction : son poids fonctionnel (colonne FW), son poids logiciel (colonne SW), le poids du développement (colonne DW), le coût du développement (colonne DC), et enfin le poids de la qualité (colonne DCQ).

Tableau 1–3 Tableau d'évaluation de la solution n°3

SOLUTION 3								
Zone	**MF**	**SF**	**Description**	**FW**	**SW**	**SDW**	**DC**	**DCQ**
PRODUCTION								
	E1	Importer						
		FP11	Importer dans la base une structure de données définie extérieurement.	1	2	3	7	8,89
		FP33	Convertir en XML les unités d'information non-XML. Convertir les illustrations aux standards retenus.	1	2	3	7	8,89
		FP31	Disposer d'un programme permettant d'introduire dans la base des unités d'information.	1	2	3	5	6,35
	E2	Créer						
		FP12	Créer des squelettes de publication.	1	1	1	1	1,27
		FP13	Créer une structure de données en XML.	1	2	3	3	3,81
GESTION								
	G1	Identifier						
		FP35	Disposer d'un programme permettant d'introduire des illustrations dans la base.	1	2	2	2	2,54
		FP41	Gérer les unités d'information et les graphiques associés.	1	1	1	1	1,27
		FP45	Gérer les chemins d'accès aux unités d'information qui seraient stockées à l'extérieur de la base.	1	1	1	1	1,27
	G2	Protéger						
		FP23	Contrôler les accès aux différentes fonctions relatives à la gestion des structures de publication.	1	2	2	0,5	0,635
		FP43	Contrôler les accès aux unités d'information (notion de workflow).	1	1	1	0,5	0,635
		FP63	Contrôler les autorisations d'accès aux fiches d'identité.	1	2	3	3	3,81
		FP93	Pouvoir mettre les publications dans une zone protégée du système.	0	0	0	0	0
	G3	Cadencer						
		FP64	Gérer les processus.	1	1	1	0,5	0,635
DIFFUSION								
	P1	Assembler						
		FP71	Produire une structure de données en une instance XML contenant l'ensemble des textes.	1	1	1	1	1,27

Tableau 1–3 Tableau d'évaluation de la solution n°3 (suite)

SOLUTION 3								
Zone	**MF**	**SF**	**Description**	**FW**	**SW**	**SDW**	**DC**	**DCQ**
		FP72	Pouvoir effectuer cet export selon différents critères de sélection.	1	2	3	3	3,81
		FP81	Pouvoir choisir parmi plusieurs formats de présentation.	1	2	3	3	3,81
	P2	Adapter						
		FP82	Disposer des convertisseurs XML->papier.	1	1	1	0	0
	P3	Visualiser						
		FP74	Visualiser des unités d'information ou des graphiques depuis l'interface de gestion de la base.	1	1	1	0	0
		FP83	Disposer d'un navigateur.	1	2	3	3	3,81
		FP84	Prévoir un poste de contrôle des versions papier ou électroniques produites.	1	1	1	1	1,27
	P4.	Transmettre						
			…	…	…	…	…	…
			…	…	…	…	…	…
			…	…	…	…	…	…
			…	…	…	…	…	…
			TOTAUX	36	53	68	85	108

En reportant ces chiffres sur une courbe, on obtient finalement une représentation dont un exemple est fourni par la figure 1-9. Les différences entre solutions apparaissent alors de manière visible et synthétique. Comparer les solutions en étudiant les raisons des différences de coût point par point devient, dès lors, d'une grande simplicité.

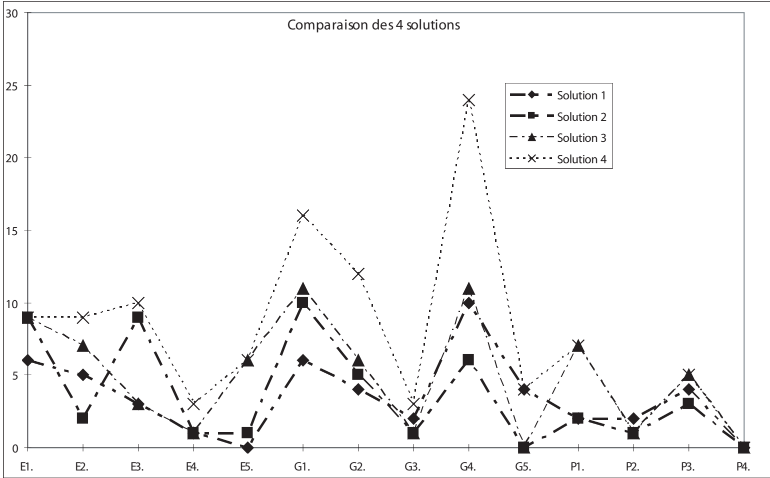


Figure 1–9 Courbes comparatives des quatre solutions de notre exemple

On retiendra pour conclure qu'une telle représentation parle tant aux techniciens qu'aux futurs utilisateurs ou décideurs. Elle est un outil de dialogue particulièrement adapté.

En résumé…

La méthode que nous avons présentée permet tout à la fois de structurer la démarche projet et de représenter de manière parlante les fonctions du futur système, tout en fournissant des indices utiles quant aux coûts respectifs des développements, au nombre de fonctions « natives », et aux expertises nécessaires au développement.

Le découpage fonctionnel donne une première idée de l'architecture des composants logiciels à mettre en place tandis que la mise en évidence des services renseigne dès le départ sur les flux de données internes et externes au système.

Cette méthode permet donc de placer un certain nombre de bases qui sont autant de fondations du futur système.

Appliquée préalablement à un choix définitif de solution, elle a l'avantage de répertorier très précisément les réelles fonctions du futur système, sans se limiter aux seules fonctions vues des utilisateurs. Les fonctions coûteuses sont rapidement identifiées. Les courbes permettent donc aux décideurs de faire des choix de mise en œuvre. Les chefs de projet ont un outil qui leur facilite les explications relatives à tel ou tel coût de développement.

Au terme de l'application de cette méthode, vous apercevez les premières relations liant les données aux fonctions du système, c'est-à-dire les grandes masses pour lesquelles des modèles de données devront être développés, cela avant même de vous être engagé dans une quelconque étude détaillée.

2

Étape 2 - Réaliser le modèle conceptuel

Toute personne ayant étudié plusieurs langues sait qu'un même fait peut être exprimé de manière assez différente d'une langue à l'autre. Chacune apporte sa propre perspective, ce qui enrichit l'analyse du fait étudié mais pose en même temps des problèmes de traduction, puisqu'il y a rarement correspondance exacte entre les différentes perspectives. Il en est de même pour les langages formels comme UML ou XML.

> **VOCABULAIRE Langage formel**
>
> Langage qui utilise un ensemble de termes et de règles syntaxiques pour communiquer sans aucune ambiguïté (par opposition à langage naturel). Arrêté du 30 décembre 1983 – *J.O.* du 19 février 1984.

Entre UML et XML, la différence principale réside dans la manière dont la notion de relation est envisagée. Les relations semblent une notion basique, qui devrait aller de soi. C'est pourtant un sujet qui demande à être analysé avec attention. En particulier, UML et XML ont une manière différente d'aborder les relations et il est essentiel d'en comprendre les différences si l'on veut étudier la correspondance entre ces deux langages.

Nous étudierons en premier lieu la différence entre les modèles hiérarchiques propres à XML et les modèles d'association propres à UML. Nous verrons ensuite comment XML et UML organisent les différents modèles qu'ils proposent en fonction de leur niveau d'abstraction. Enfin, nous nous intéresserons au rôle de chacun de ces modèles dans les différentes étapes de conception de système de traitement de l'information.

Différence entre modèle hiérarchique et modèle d'association

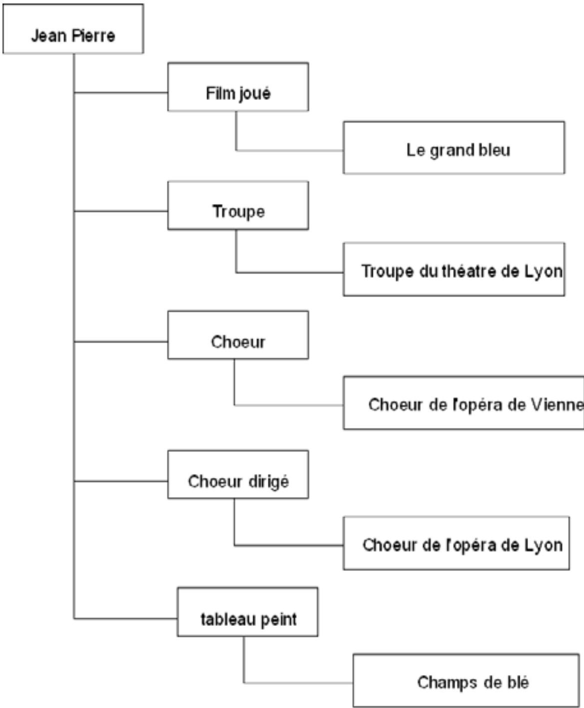
XML exprime l'information sous forme de relations hiérarchiques constituées d'objets emboîtés les uns dans les autres à la manière des poupées russes. De son côté, UML divise l'analyse de l'information en deux catégories ; d'un coté les objets et d'un autre coté les relations qui les unissent. Pour UML, les relations entre les objets ne sont pas systématiquement hiérarchiques. Ce qui importe, pour UML, est la manière dont les objets sont impliqués dans les relations ; on dit aussi la manière dont ils participent aux relations. Pour décrire cette participation, UML utilise la notion de rôle. Dans l'exemple que nous prenons ci-après, `Jean` a cinq rôles : *acteur*, *membre d'une troupe*, *choriste*, *chef de chœur* et *peintre*. Ces rôles vont de pair avec ceux des autres objets auxquels Jean est associé : *troupe*, *film joué*, *chœur*, *chœur dirigé* et *tableau peint*.

Dans la figure 2-1, notre exemple est présenté sous une forme uniquement hiérarchique. L'emboîtement hiérarchique ne fait apparaître qu'une partie des rôles évoqués précédemment. Dans la figure 2-2, les relations sont au contraire plus explicites. Dans cette seconde représentation, les relations sont sur le même plan que les objets : c'est une caractéristique du modèle UML. Le mot précisément utilisé dans UML pour parler de ces relations est *association*.

Exemple de représentation en XML n'utilisant que les mécanismes d'emboîtement d'éléments pour simuler une stricte hiérarchie

```
<artiste nom="Jean">
  <film_joue nom="Le grand bleu" />
  <troupe nom="Troupe du théatre de Lyon"/>
  <Choeur nom="Choeur de l'opéra de Vienne" />
  <Choeur_dirige nom="Choeur de l'opéra de Lyon" />
  <Tableau nom="Champs de blé"  />
</artiste>
```

Figure 2–1
Modèle centré sur la hiérarchie



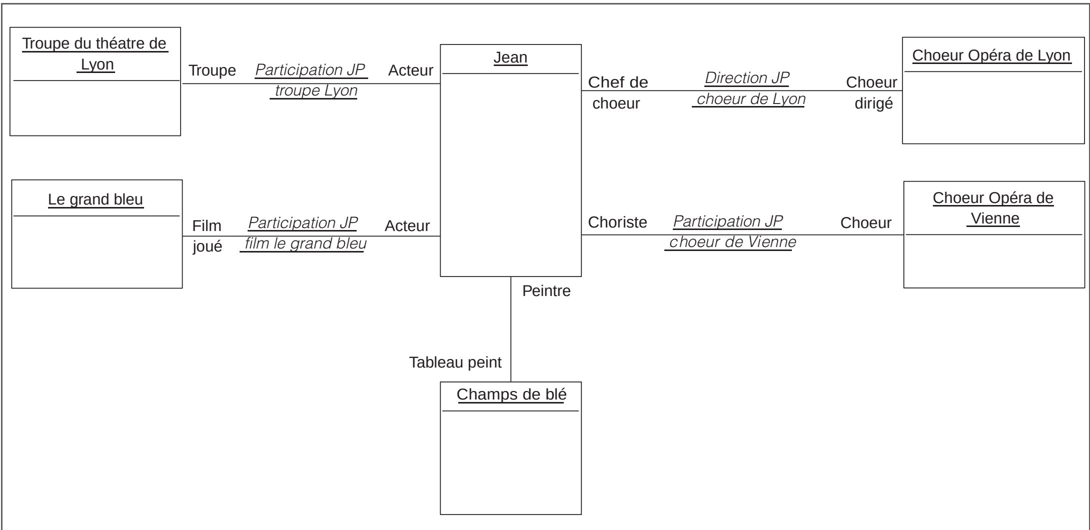


Figure 2–2 Modèle centré sur les associations

Les emboîtements XML ont cet avantage (qui est en même temps une limitation) de proposer *de facto* au moins deux relations (la relation parent-enfant et la relation de fratrie) qui suffisent à définir un sens de lecture aux données et confèrent à l'information une organisation aux contours bien délimités. La relation parent-enfant de XML se traduit très concrètement par les emboîtements bien connus des balises. Dans l'exemple précédent, le modèle hiérarchique se lit en partant de l'artiste `Jean` puis, suivant les balises, en découvrant les films joués, troupes de théâtre, chœurs d'opéra etc. Dans le modèle objet/association, il n'y a pas véritablement de sens de lecture. Cette dernière peut aussi bien commencer par `Jean` ou par `Le grand bleu`. Cette absence de sens de lecture vient du fait que les associations sont mises sur le même plan que les objets. L'objectif premier est de mettre en évidence la raison d'être des associations qui unissent les objets, *via* les rôles. Ils donnent très clairement, pour chacun des objets *participants* de l'association, la raison pour laquelle ils sont associés : `chef de chœur` pour `Jean` et `chœur dirigé` pour `Chœur de l'opéra de Lyon`.

On perçoit dès maintenant les rôles complémentaires de UML et de XML : UML met en évidence ce qui est fondamental dans un modèle de données – à savoir les associations qui unissent les données entre elles et leur raison d'être – tandis que XML fournit aux programmes un sens de lecture structuré de ces données qui permettra de les exploiter facilement.

Dans la suite de cet ouvrage, nous utiliserons la notation UML pour représenter les modèles de données sous l'angle des relations et XML pour montrer une implémentation concrète de ces relations.

Modèle d'objet UML et document XML

Dans notre modèle exemple précédent, nous avons utilisé des mots correspondant aux objets et personnes du monde réel : `Jean`, `Chœur del'opéra de Lyon`, `film joué`, `chœur dirigé`, etc. Le modèle correspondant s'appelle *modèle d'objets* dans UML et tout simplement *document* dans XML. L'étape suivante consiste à identifier les caractéristiques communes à tous les objets étudiés pour les ranger dans des classes. Les modèles représentant les caractéristiques communes aux objets s'appellent modèles de classes en UML et schémas XML en XML.

Dans les paragraphes suivants, nous allons présenter le modèle d'objets UML et le modèle des documents XML. Les modèles de classes feront l'objet de la section suivante.

Modèle d'objets UML

Le modèle d'objets UML est un graphe représentant des objets individuels et les associations qui les unissent. Il est souvent utile de produire un modèle d'objets à titre d'exemple pour étudier des cas concrets avant l'étude de leurs caractéristiques communes faite dans le modèle de classes.

Une fois que le modèle de classes est disponible, on peut revenir sur le modèle d'objets pour y indiquer la classe dont chaque objet est une instance. Dans la figure 2-3, la notation utilisée fait apparaître pour chaque objet son nom complété du nom de sa classe (les deux noms sont séparés par le caractère deux-points). On peut ainsi lire que `Jean` est une instance de la classe `Artiste`.

> **VOCABULAIRE Faut-il dire instance ou occurrence ?**
>
> La traduction française correcte du mot anglais *instance* est occurrence, et l'expression « modèle d'occurrences » est celle utilisée par les éditeurs dans les logiciels. Cependant, dans les communautés UML et SGML, le terme d'instance est couramment usité. C'est pourquoi nous l'avons conservé ici.

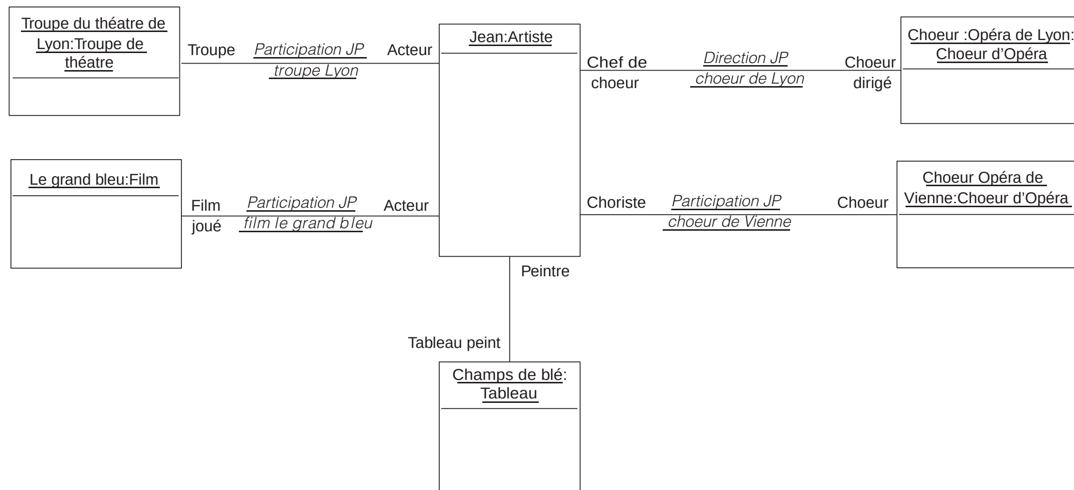


Figure 2–3 Modèle d'objets UML avec référence aux classes des objets

Modèle de base des documents XML : Infoset

De la même manière que l'on peut faire un modèle d'objets UML indépendamment de tout modèle de classes, on peut bien évidemment faire un document XML indépendamment de tout schéma ou de toute DTD ; cela est d'ailleurs une caractéris-

tique fondamentale de XML, comme nous l'avons rappelé dans l'introduction. C'est le modèle Infoset qui indique comment tout document XML doit être constitué, indépendamment de tous DTD et schéma XML.

REMARQUE La recommandation Infoset

Infoset est une recommandation du W3C du 24 octobre 2001 disponible à l'adresse suivante :
http://www.w3.org/TR/2001/REC-xml-infoset-20011024/

D'après la recommandation XML Infoset, un document XML est constitué de onze *unités d'information*. Chacune d'entre elles est caractérisée par un ensemble de propriétés et est définie indépendamment de la syntaxe XML proprement dite (balise, balise de début, balise de fin…). En effet, il est admis aujourd'hui que la représentation physique la plus répandue des documents XML – faite des signes < et > et des balises de début et fin – n'en est qu'une des formes possibles. Quand un document XML est chargé en mémoire ou est stocké dans une base de données, sa forme diffère.

Ces unités d'information peuvent être décrites de plusieurs manières :

- sous la forme de descriptions textuelles comme le fait la recommandation ;
- sous la forme de schéma RDF comme le propose la recommandation dans ses annexes ;
- Enfin, sous la forme d'un modèle UML comme nous l'avons fait à la figure 2-4.

TECHNIQUE XML : RDF

RDF – pour *Resource Description Framework* – est une spécification du W3C pour décrire les métadonnées visant à offrir un langage de description de ressources Web compréhensible par les ordinateurs et les humains. Ce projet porte aussi le nom de « Web sémantique ». Bien qu'un ensemble de métadonnées RDF s'écrive sous la forme d'un document XML, RDF n'utilise ni les DTD ni les schémas XML pour en spécifier le modèle, mais une technique qui lui est propre. La spécification RDF est disponible sur le site du W3C à l'adresse suivante : http://www.w3.org/RDF.

Nous fournissons également au tableau 2-1 un résumé en texte naturel des définitions de ces onze objets, dont la présence dans les documents XML est autorisée par Infoset. Par ailleurs, nous en donnons une description plus précise en annexe de cet ouvrage.

Infoset fournit ainsi le modèle de tous les documents XML. D'un point de vue hiérarchique, les unités d'information les plus significatives qui s'en dégagent sont le document, les éléments et les attributs.

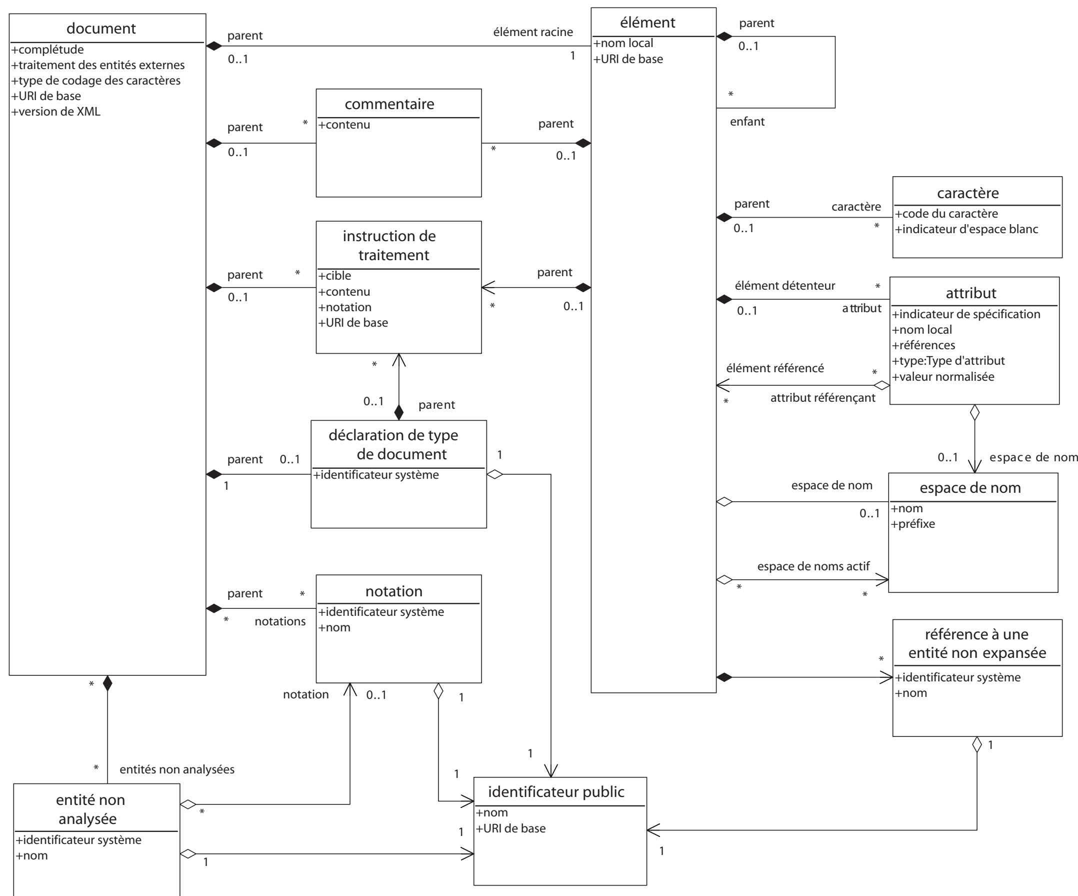


Figure 2–4 Modèle conceptuel d'un document XML selon la recommandation Infoset

Tableau 2–1 Description résumée des unités d'information d'Infoset

Unité d'information	Propriétés

Document	L'unité d'information document est celle par laquelle commence le document XML. Souvent, elle est appelée prologue.
	
	Exemple : ci-après trois lignes composées de deux instructions de traitement et d'une déclaration de type de document qui sont les propriétés d'une unité d'information de type document.
	`<?xml version='1.0' ?>`
	`<?xml:stylesheet type="text/xsl" href="sb.xsl"?>`
	`<!DOCTYPE SB SYSTEM "sb.dtd" []>`

Tableau 2–1 Description résumée des unités d'information d'Infoset (suite)

Unité d'information	Propriétés
Élément	Les unités d'information de type élément correspondent aux balises contenues dans un document XML. En particulier, l'élément racine est une unité d'information de type élément qui fait l'objet d'une propriété de l'unité d'information de type document. Exemple : <body>
Attribut	Ces unités d'information correspondent aux attributs portés par les éléments. Elles sont utilisées pour tous les attributs, y compris ceux définis avec une valeur par défaut dans la DTD, ainsi que l'attribut spécial de déclaration d'espace de noms. Exemple : <plansect sectname="eff">
Instruction de traitement	Il s'agit des instructions de traitement présentes dans le document XML. Exemple : <?xml version='1.0' ?>
Référence d'entité non résolue	L'unité d'information de type référence d'entité non résolue est un marqueur par lequel le processeur XML indique qu'il n'a pas expansé l'entité. La définition de l'entité ainsi que son utilisation forment une seule unité d'information, par exemple : dans la DTD : < !ENTITY fig422 SYSTEM "c:/user/graphiques/image.tif"> et dans le document XML : <figure file="**fig422**"/>
Caractère	Il existe une unité d'information de ce type par caractère de donnée du document, même ceux qui apparaissent sous la forme d'une codification, tels les caractères accentués. Exemple : bla bla bla m≖me Dans la deuxième ligne, l'appel d'entité ≖ représente le caractère ê et est donc considéré comme étant un appel d'unité d'information de type caractère.
Commentaire	Il existe une unité d'information de ce type par commentaire présent dans le document, sauf pour les commentaires déjà inclus dans la DTD du document. Exemple : <!-- ceci est un commentaire -->
Déclaration de document	Cette unité d'information existe quand le document XML a une déclaration de type de document. Les éventuelles déclarations d'entités et notations définies dans la déclaration de type de document sont des propriétés de l'unité d'information de type document et non du document dans lequel la déclaration est faite. Exemple : <!DOCTYPE SB SYSTEM "sb.dtd" [<!NOTATION cgm PUBLIC "-//USA-DOD//NOTATION Computer Graphics Metafile//EN">]>

Tableau 2–1 Description résumée des unités d'information d'Infoset (suite)

Unité d'information	Propriétés

Entité non analysée	Une unité d'information de type entité non analysée existe pour chaque entité générale déclarée dans la DTD.
	Exemple :
	`<p>&sprsede2;</p>`
	L'entité générale `sprsede2` est définie dans la DTD, par exemple de la manière suivante :
	`<!ENTITY sprsede2 "Originator Supplies Superseded Publication Number Here">`

Notation	Une unité d'information de type notation existe pour chaque notation définie dans la DTD. Exemple : `<!NOTATION cgm PUBLIC "-//USA-DOD//NOTATION Computer Graphics Metafile//EN">`
Espace de noms	Chaque élément du document a comme propriété une unité d'information de ce type pour chaque espace de nom en vigueur au niveau de l'élément. Exemple : `<xsd:element xmlns:xsd="http://www.w3.org/2001/XMLSchema">`

Notion de classe

À partir d'objets du monde réel, comme ceux de notre exemple précédent – Jean, *Le Grand Bleu* –, il faut pouvoir exprimer des notions plus génériques telles que :

- `Film` au lieu de `Le Grand Bleu`,
- `Artiste` au lieu de `Jean`,
- `Tableau` au lieu de `Champs de blé`.

Par ce processus d'abstraction, on définit alors des groupes d'objets ayant tous les mêmes caractéristiques : cela s'appelle une classe. Les objets d'une classe ont :

- le même jeu de caractéristiques communes : les propriétés ;
- les mêmes associations aux autres objets : celles définies pour la classe.

UML permet de définir une classe au travers de ses propriétés et de ses associations avec d'autres classes.

Via les DTD ou XML Schema, XML définit des classes au travers de propriétés (les attributs des éléments) et des relations d'emboîtement qui lient les éléments entre eux (relations parent-enfant et relations de fratrie).

XML Schema a apporté une nouveauté importante : la définition d'une classe (un type) est dissociée de l'attribution d'un nom d'élément. Ainsi, comme avec les DTD, si différents noms d'éléments peuvent représenter une même classe, on peut égale-

ment avoir un même nom d'élément représentant différentes classes en fonction de l'endroit où il est utilisé dans le document. Ce mécanisme est similaire à celui que jouent les noms de rôle dans les associations des modèles UML, à ceci près que les éléments XML définissent des relations d'emboîtement alors que les associations UML ne sont pas systématiquement hiérarchiques.

Modèles et métamodèles

Dans cet ouvrage, nous nous concentrons sur les modèles permettant de décrire les caractéristiques et structures communes que doivent respecter les données et documents. C'est d'une part le modèle de classe UML et d'autre part les différents types de schémas XML que sont les DTD, XML Schéma, RelaxNG, etc... Dans cette section cependant, nous allons faire un aperçu rapide des différents niveaux d'abstraction qui permettent de décrire des modèles à partir d'autres modèles. On appelle ces modèles de modèles des métamodèles. On obtient alors un emboîtement de modèles correspondant chacun à un niveau d'abstraction différent

> **VOCABULAIRE méta**
>
> Le terme de méta est utilisé dans métamodèle et métalangage. Dans les deux cas, sa signification est la même. Un métamodèle est un modèle qui sert à décrire d'autres modèles et un métalangage est un langage qui sert à décrire d'autres langages.

Ainsi passe-t-on du modèle d'objet décrivant `Jean-Pierre` en tant qu'`Artiste` au modèle de classe décrivant les caractéristiques d'un artiste en général, puis au modèle décrivant les concepts dont on se sert dans un modèle de classe (classe, association, …).

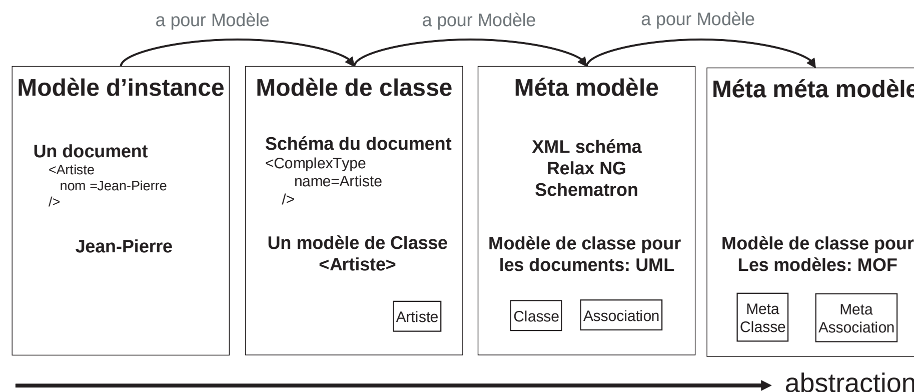


Figure 2–5 Modèles et métamodèles

Le principe d'abstraction de modèle à métamodèle s'applique à XML et UML aussi bien qu'à d'autres langages de modélisation. Il existe cependant une différence d'objectif entre XML et l'ensemble des techniques de modélisation associées à UML. Ce dernier fait partie d'une famille de métamodèles gérée par l'OMG : Object Management Group. Pour gérer cette famille, il est nécessaire d'avoir un autre modèle permettant de décrire tous les métamodèles. C'est le rôle du MOF (Meta Object Facility) que l'on considère aussi comme un méta-métamodèle. Le MOF apparaît en fin de la chaîne d'abstraction dans la figure 2-5.

XML n'est pas un langage dédié à la modélisation et, de manière opérationnelle, n'a pas pour vocation de sensibiliser les concepteurs à la distinction entre métamodèle et méta-métamodèle. Aussi n'y retrouve-t-on pas la distinction effectuée dans UML entre classe et métaclasse. En revanche, XML fait la distinction entre un document et son modèle de données appelé plus généralement schéma structurel. Il existe d'ailleurs plusieurs types de schémas structurels pour XML et les plus répandus sont les DTD, XML Schema, RelaxNG et Schematron. Bien que la forme finale d'un document XML soit toujours la même, les règles qui en spécifient les structures peuvent être édictées de différentes manières. Les unes vont s'attacher à définir des modèles de contenu (c'est l'approche DTD et RelaxNG), d'autres des types (c'est l'approche XML Schema), d'autres enfin vont s'attacher à définir des règles relatives à la valeur même des données (c'est l'approche Schematron). Ainsi, sans être complètement opposables (on peut parfaitement définir des structures avec Schematron par exemple), toutes les techniques de modélisation officielles ont leur singularité.

XML dispose en outre d'un modèle pour les documents bien formés – Infoset – qui s'applique à tout type de document XML. Ainsi, lorsque les schémas XML sont eux-mêmes exprimés en XML, ce qui est le cas d'XML Schéma et de RelaxNG, on peut contrôler ou accéder à ces schémas à l'aide du modèle Infoset.

On peut illustrer notre propos par la figure 2-6 :

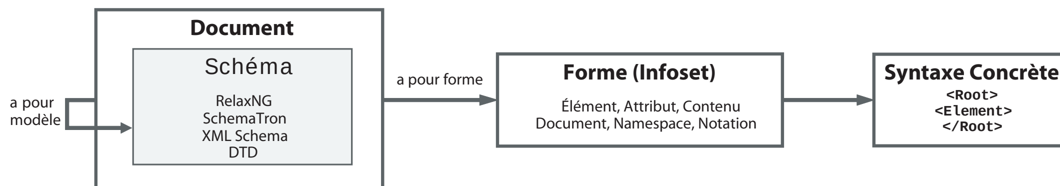


Figure 2–6 Modèle de forme et schéma

Modèles de données conceptuels, logiques et physiques

Les premiers modèles de données à réaliser dans l'étude d'une application sont ceux qui décrivent les concepts mis en jeu dans le domaine étudié, indépendamment de toute implémentation ou contrainte technique relative en particulier à la question du stockage des données. Ces modèles doivent aussi être indépendants des langages utilisés pour traduire concrètement la structure de l'information : en particulier de XML Schema, RelaxNG et Schematron. Pour ces raisons importantes, ces modèles sont dits conceptuels.

Les modèles conceptuels mettent en œuvre des concepts, définissent des vocabulaires, donnent des limites et des caractéristiques aux objets et fixent les relations qu'ont les objets entre eux. Ils doivent pouvoir être exprimés simplement. Leur audience est a priori plus large que celle des experts informatiques puisqu'ils doivent aussi être lus par des experts métier. Le diagramme de classes/associations d'UML est particulièrement bien adapté pour représenter des modèles conceptuels. La notation graphique offerte par le diagramme de classes donne une vue synthétique des concepts et de leurs relations. Cette forme de représentation est connue par une large communauté de concepteurs et d'analystes métier, ce qui permet une meilleure communication entre les équipes projets.

Les modèles logiques ont pour objet la prise en compte de l'architecture informatique dans laquelle s'intégreront les données. C'est à ce niveau que sont définis les principes de structuration des données. Leur articulation en îlot de données se base sur les principes d'urbanisation du système et du découpage de ce dernier en services comme cela a été vu au chapitre 1. Pour les modèles logiques, on utilise tantôt le formalisme UML, tantôt le formalisme XML, selon l'expertise des acteurs du projet. C'est pour ce type de modèle que l'on retrouve le plus souvent les débats sur la correspondance UML/XML. Ces derniers feront l'objet du chapitre suivant.

Enfin, les modèles physiques prennent en compte les syntaxes concrètes qui seront utilisées pour capturer les données : des fichiers plats, une base de données relationnelle ou XML, la validation par XML Schema ou RelaxNG, etc.

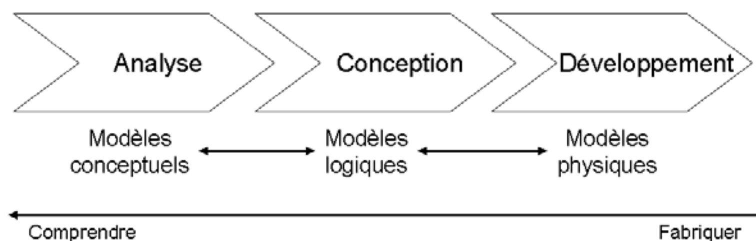


Figure 2–7
Modèles et démarche de conception

L'usage des différents types de modèles, conceptuels, logiques ou physiques s'inscrit le plus souvent dans une démarche de projet. De façon très classique, on distingue les étapes d'analyse, de conception et de développement. De manière tout aussi classique, on associe les modèles conceptuels à la phase d'analyse, les modèles logiques à la phase de conception et les modèles physiques à la phase de développement (voir figure 2-7).

Cependant, deux risques d'écueils majeurs doivent être évités lorsque l'on aborde les modèles de données sous l'angle de la démarche de projet :

- Confondre le passage des modèles conceptuels aux modèles physiques avec le passage d'une description simple – le modèle conceptuel – à une description complète et détaillée – le modèle physique.
- Suivre à la lettre dans chaque projet la distinction entre modèles conceptuels, logiques et physiques.

Lorsque le cas étudié est simple, il n'est pas utile de faire la distinction entre les trois types de modèles. Par exemple, si le cas étudié concerne un artiste et les tableaux qu'il a peints en vue d'un affichage à l'écran, un simple fichier XML doit suffire pour représenter les concepts, le déploiement et le stockage. Si, au contraire, on cherche à concevoir l'ensemble des portefeuilles de titres proposés par une banque en prenant en compte les différents canaux de distribution, il sera indispensable de distinguer les différents types de modèles lors de l'étude.

Enfin, la distinction entre modèles conceptuels, logiques et physiques ne réside pas dans le niveau de détail de chacun. On peut déterminer des transformations et des correspondances entre les différents modèles, mais il ne s'agit pas de « niveau de zoom » permettant de progresser du général au particulier.

On retiendra que les modèles de conception, parce qu'ils ont pour objectif principal d'identifier les associations entres les concepts, feront plus largement usage d'UML. Pour la structuration des données et leur matérialisation physique, XML est le candidat naturel parce qu'il peut jouer le rôle de format d'implémentation et parce qu'il propose intrinsèquement des principes de structuration des données.

Le cas PiloteWeb

Présentation du cas

L'AAF – Association des aéroclubs de France – a pour objet de promouvoir les relations entre pilotes amateurs afin de leur faire découvrir les possibilités de voyage à titre privé en utilisant le réseau des aérodromes en France. L'association a décidé de

mettre à disposition sur Internet une application permettant aux pilotes d'établir des plans de vols à partir d'une description du réseau d'aérodromes et des facilités d'hébergement et de déplacement qui leur sont accordées.

L'application est conçue à la manière des pilotes qui se repèrent en fonction des sites géographiques. Chaque site est caractérisé par un nom, une photo et une position définie en latitude et longitude.

Outre leurs caractéristiques de sites, les aérodromes ont un code et une fréquence radio. Pour chaque aérodrome, on dispose des conditions météorologiques suivantes : la vitesse du vent, la température.

Une liste de partenaires est associée à chaque aérodrome. Outre leurs caractéristiques de site, les partenaires ont une adresse et un numéro de téléphone. Les partenaires peuvent être des aéroclubs, des loueurs de voitures ou des restaurants. Les restaurants précisent leurs horaires d'ouverture pour les repas de midi et du soir.

Les pilotes sont identifiés par leur adresse électronique et un mot de passe. Ils ont aussi une position géographique qui permet de retrouver les pilotes présents sur le même aérodrome.

L'application PiloteWeb permet aux pilotes enregistrés d'établir des plans de vols. Un plan comprend plusieurs étapes. À chaque étape, l'application présente l'aérodrome le plus proche, ainsi que la liste des partenaires de l'aérodrome et la liste des pilotes présents sur la zone.

Modèle conceptuel de données de PiloteWeb

La construction d'un modèle conceptuel de données se déroule généralement en trois grandes phases : découverte des principaux concepts, mise au jour des principales relations, recherche et formalisation des concepts et relations intermédiaires.

Découverte des principaux concepts

La première phase de construction d'un modèle conceptuel de données consiste tout simplement à lister les principaux concepts tels qu'ils apparaissent à la première lecture du cas étudié. Pour PiloteWeb, on peut distinguer clairement les concepts de pilote, d'aérodrome, de site, de partenaire, d'aéroclub, de loueur de voitures et de restaurant. Les concepts sont représentés par des classes en UML. De même, certaines caractéristiques des concepts identifiés apparaissent à la simple lecture du cas. Il en est ainsi du nom du pilote, de son adresse électronique et de son mot de passe ; un partenaire a une adresse et un numéro de téléphone. Ces caractéristiques sont représentées par des attributs de classe en UML.

Dans cette première étape de découverte, on s'attachera à donner une définition précise à chaque classe. Par exemple, la classe `site` décrit un lieu géographique et non pas un site Internet.

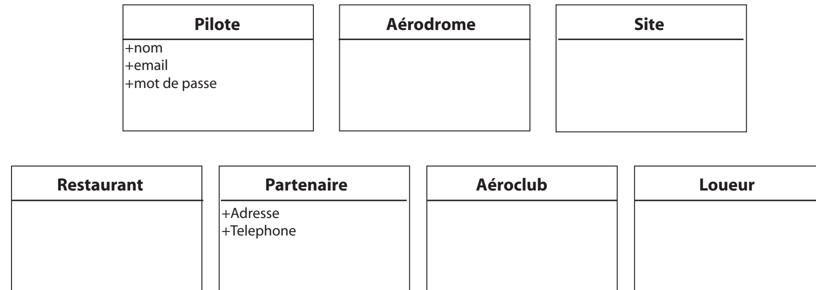


Figure 2–8
Classes de PiloteWeb

Découverte des principales relations

Dans un deuxième temps, il faut faire apparaître les relations entre les classes. On distingue deux types de relations. Les relations de sous-typage et les relations associatives ou associations. Les relations de sous-typage permettent de capitaliser des caractéristiques communes à certains concepts. Dans le cas de PiloteWeb, la classe `Partenaire` répond à ce critère de capitalisation. Les restaurants, les aéroclubs et les loueurs sont tous des cas particuliers de partenaires. Ils sont donc sous-types de la classe `Partenaire`.

Dans UML, les relations de sous-typage sont représentées par une flèche allant du sous-type vers son sur-type. Il est possible de combiner les différentes flèches de relation de sous-types pour alléger le diagramme.

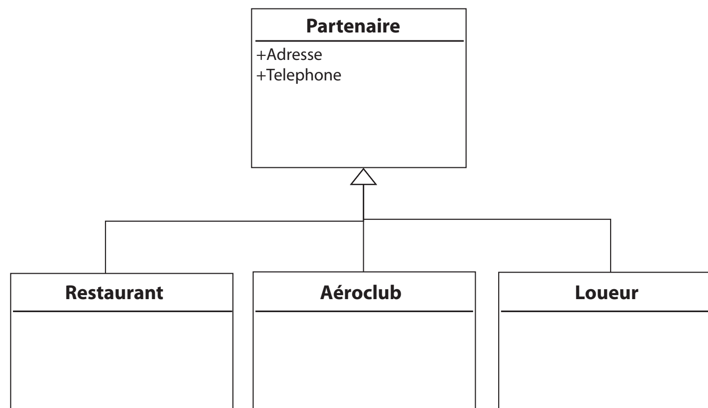


Figure 2–9
Classes sous-types de Partenaire

Les associations indiquent un rapport existant entre deux classes. Un rôle indique, pour chaque classe, sa place dans l'association. Dans le cas de PiloteWeb, les pilotes, les aérodromes et les partenaires ont une association avec un site indiquant leur localisation. Un site joue donc les rôles de `localisation pilote`, `localisation aérodrome` ou encore `localisation partenaire`.

De manière optimale, chaque rôle devrait avoir un nom. Ainsi, au rôle `localisation pilote` correspond le rôle `pilote localisé`. Cependant, l'usage veut que, lorsque l'association n'est pas ambiguë, le nom donné au rôle soit celui de la classe qu'il représente. Dans ce cas, le nom du rôle est omis dans le diagramme de classe. C'est ce qui a été retenu dans l'exemple PiloteWeb pour les rôles d'`Aérodrome` et de `Partenaire` dans les associations avec la classe `Site`.

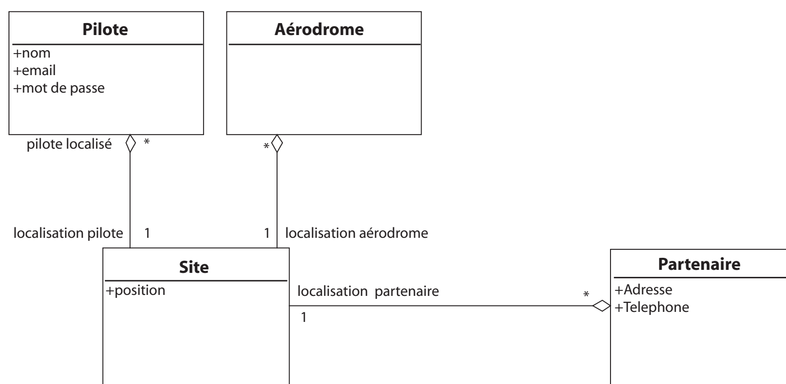


Figure 2–10
Associations

La recherche des noms de rôle est souvent un exercice délicat. C'est pourtant un excellent moyen d'éviter les approximations dans la conception des modèles de données. Supposons, dans notre exemple PiloteWeb, que nous n'ayons pas donné les noms de rôle `pilote localisé` et `localisation pilote` à l'association entre `Pilote` et `Site`. On ajoute maintenant une nouvelle association définissant les rôles de `pilote responsable` et `site géré`. La simple lecture de la figure 2-11 montre que l'on ne sait plus très bien ce que signifie l'association sans nom de rôle entre `Pilote` et `Site`. On devine que la classe `Pilote` a une association avec la classe `Site`, mais on ne sait plus pourquoi.

Les noms de rôle sont donc porteurs de sens. On parle de sémantique de l'association. L'absence de nom de rôle laisse une liberté d'appréciation de la signification de l'association. Cela peut conduire à des erreurs d'interprétation du modèle conceptuel lors du passage en phase de réalisation.

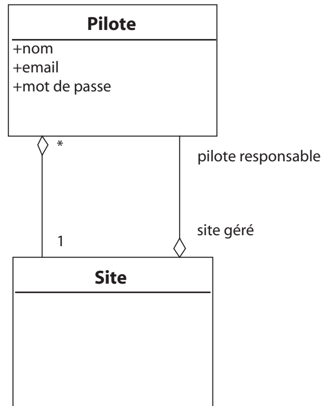


Figure 2–11
Importance des noms de rôle

Pour de nombreuses associations du modèle, nous avons utilisé la symbolique UML du losange. Dans UML, le losange blanc indique une agrégation. L'agrégation est une association type tout/partie exprimant une subordination faible entre deux classes telle que l'indique la figure 2-12. Le rôle représentant *le tout* est le rôle d'agrégation. Il désigne la classe « subordonnante » dans l'association. L'autre rôle représente *les parties* ; c'est le rôle de subordination. Il indique la classe subordonnée dans l'association.

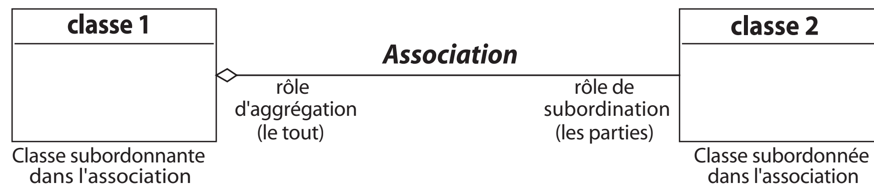


Figure 2–12
Agrégation

Si l'on retire une des parties d'un tout, ce dernier s'en trouve modifié. Par exemple, pour l'association entre `Pilote` et `Site` (voir figure 2-10), le rôle `pilote localisé` indique que la classe `Pilote` est l'agrégat. Si l'on retire cette association, la définition de `Pilote` s'en trouve modifiée alors que celle de `Site` reste inchangée. Pour savoir si un des rôles d'une association est un agrégat, il faut donc se poser la question suivante : y a-t-il une classe dont la définition se trouve modifiée par l'ajout ou la suppression de l'association ?

L'intérêt de repérer les agrégations est de permettre une première identification du responsable de l'association. Nous verrons dans le chapitre sur les modèles logiques que cette information est utile pour les choix d'implémentation des associations dans les modèles physiques XML.

Outre l'agrégation simple, le modèle de classe UML comprend un autre type d'agrégation appelé « composition ». Les compositions sont représentées par des losanges noirs. Elles indiquent que les parties sont gérées hiérarchiquement par rapport à l'agrégat : toute suppression de l'agrégat entraîne la suppression de ses parties. Dans l'exemple de PiloteWeb, toute suppression d'un site entraîne la suppression de sa photo. On dit de la composition qu'elle est une agrégation forte.

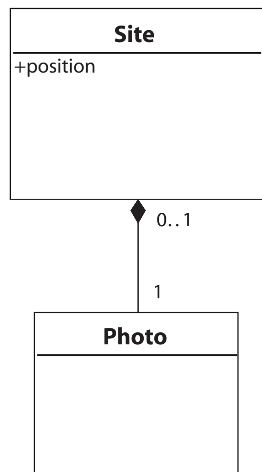


Figure 2–13
Agrégation forte : la composition

La spécification d'une composition dans un modèle conceptuel est une indication que l'association sera probablement gérée par un emboîtement parent/enfant dans l'implémentation physique du modèle en XML.

Recherche et formalisation des concepts et relations intermédiaires

Certains concepts et relations ne se déduisent pas de la simple lecture du cas à analyser. Il faut faire un effort d'analyse et des choix de modélisation, à savoir :

1. Déterminer si une donnée donne lieu à une classe ou à un attribut.
2. Déterminer si une donnée donne lieu à une classe ou à une association.

Le cas des horaires de restaurant de PiloteWeb est un bon exercice. Reprenons l'énoncé du cas : « Les restaurants précisent leurs horaires d'ouverture pour les repas de midi et du soir. » Une première proposition de modélisation pourrait être représentée par la figure 2-14. Il apparaît cependant que ce choix de modélisation mélange la représentation d'une classe – la classe `Restaurant` – et celle d'un type de données – `Heure`. Dans UML, un type de données (ou *data type* en anglais) ne peut pas avoir d'association. Le modèle de la figure 2-14 est donc invalide.

VOCABULAIRE Type de données

Un type de données définit les valeurs que peut prendre une donnée. Les type de données de base sont par exemple `entier`, `caractère`.
UML permet de définir des types de données structurés sous forme d'arborescence d'attributs. Par exemple, le type `adresse` est composé de `rue`, `ville` et `code postal`.

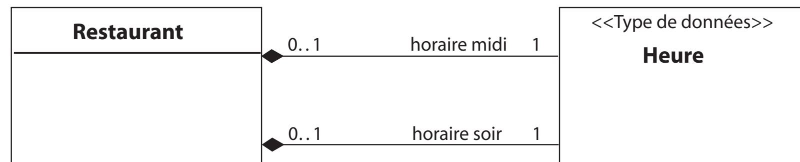


Figure 2–14 Modèle invalide ! Les types de données ne peuvent pas avoir d'association

Le modèle corrigé est fourni par la figure 2-15. Le choix de modélisation consiste à représenter les horaires comme des attributs de restaurant, attributs dont le type de données est `heure`.

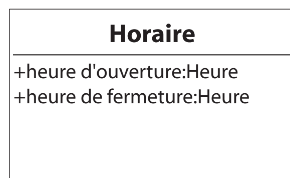


Figure 2–15 Attribut avec type de données

Cependant, le modèle de la figure 2-15 n'est pas encore satisfaisant. En effet, un horaire, ce n'est pas seulement une heure mais une heure d'ouverture et une heure de fermeture. Il nous faut donc faire émerger la classe `Horaire` en tant que telle avec un attribut `heure d'ouverture` et un autre attribut `heure de fermeture`. La classe `Horaire` a ensuite deux associations avec `Restaurant` où elle joue les rôles `horaire midi` et `horaire soir` tel que représenté à la figure 2-16.

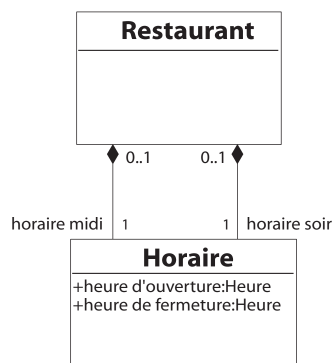


Figure 2–16 Horaire en tant que classe

L'analyse du cas pourrait se satisfaire de cet état du modèle. Cependant, un doute subsiste encore : est-ce que `Horaire` est vraiment une classe ou seulement un type de données ? On ne peut pas donner de réponse absolue à cette question. C'est un choix de modélisation qui se pose dans les termes suivants :

- Veut-on gérer des listes d'horaires indépendantes et réutilisables ?
- Les horaires ont-il des associations avec d'autres concepts ?

En cas de réponse affirmative à l'une ou l'autre de ces deux questions, il faudra considérer `Horaire` comme une classe. Dans le cas contraire, on considérera `Horaire` comme un simple type de données. C'est cette dernière solution que nous retiendrons pour l'application PiloteWeb dont la vocation première n'est pas de gérer des horaires. On obtient ainsi le modèle conceptuel de données de PiloteWeb, présenté à la figure 2-17 :

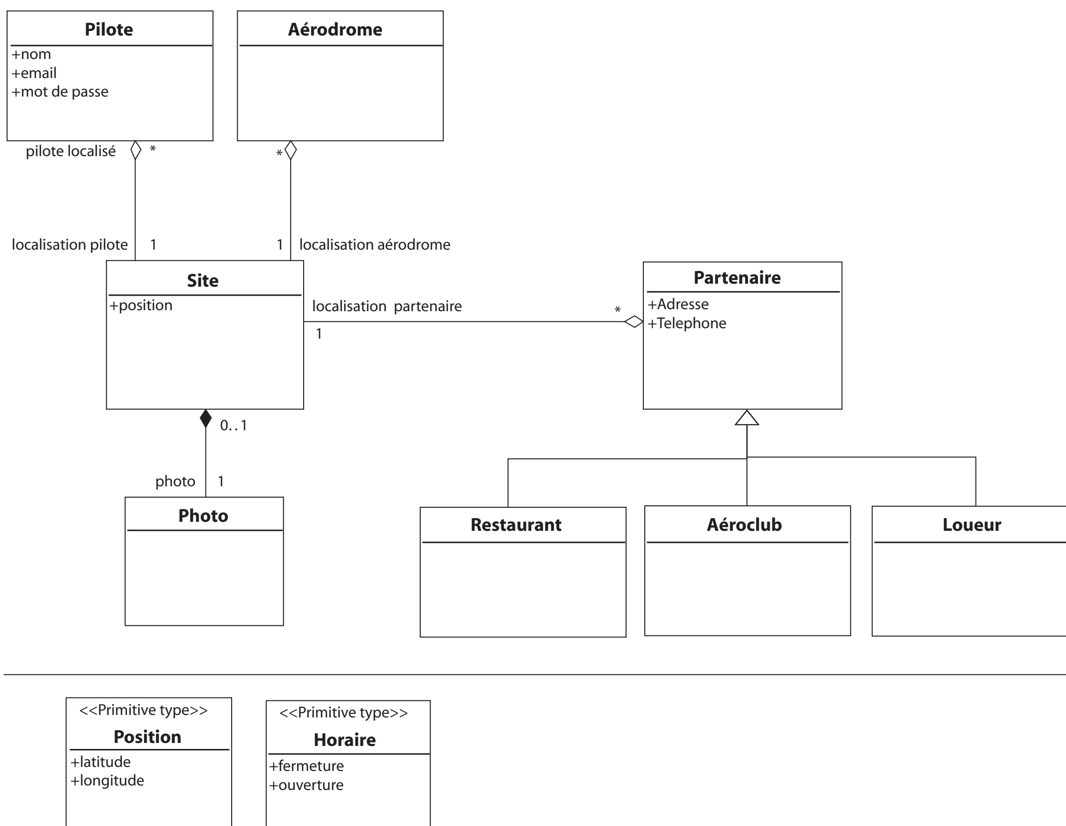


Figure 2–17 Modèle de données conceptuel de PiloteWeb

Synthèse

La recherche des concepts et relations intermédiaires nous a permis de découvrir quelques règles essentielles pour la modélisation de données :

1 Un concept est représenté par une classe. Une classe est caractérisée par ses attributs et les rôles dans les associations auxquelles elle participe.

2 Pour différencier une classe d'un type de données, il faut se poser les questions suivantes :

   – Veut-on gérer ces données sous forme d'objets indépendants et réutilisables ?

   – Les données représentées devront-elles avoir des associations avec d'autres données ?

   Une réponse affirmative à l'une ou l'autre de ces deux questions indique que la donnée doit être représentée par une classe. Dans le cas contraire, on a vraisemblablement affaire à un type de données.

3 Nommer les rôles des associations permet de préciser la raison d'être de l'association, et c'est un gage de clarté et de précision du modèle.

4 Dans de nombreux cas, on peut distinguer les associations pour lesquelles une des classes joue le rôle de tout (agrégat). La notion d'agrégation d'UML permet de caractériser de telles associations. La découverte des agrégations est une indication importante lors de la transformation du modèle conceptuel en modèle XML.

Il faut bien observer qu'il y a toujours une part de subjectivité dans les choix de modélisation. Il serait vain de penser que la simple application des règles que nous venons d'énoncer permet automatiquement de trouver le bon modèle. Dans PiloteWeb, par exemple, nous avons choisi de représenter les photos comme une classe. Cependant, si l'on applique la règle de distinction classe/type de données, elles devraient être uniquement un attribut de site avec un type de données `Photo` : en effet, PiloteWeb n'est pas une gestion d'album de photos. Cependant, l'expérience courante nous fait percevoir les photos comme des objets à part, et nous sommes naturellement enclins à les représenter sous la forme d'une classe `Photo`. Le choix d'implémentation se fera au niveau des modèles logiques.

En résumé…

Nous avons mis en avant la différence d'approche entre UML et XML : l'un représente un modèle de données basé sur des relations associatives, l'autre sur la hiérarchie des relations entre les données.

Les deux approches se complètent mais peuvent poser un problème de mise en correspondance si l'on pense pouvoir générer automatiquement un schéma XML à partir d'un modèle UML. Notre conseil est de combiner intelligemment les deux approches, en privilégiant l'emploi du modèle de classe d'UML pour l'analyse conceptuelle des modèles de données et celui d'XML pour la conception des modèles physiques.

Au travers du cas PiloteWeb, nous avons pu analyser la représentation conceptuelle d'un modèle de données et préciser la méthode permettant de découvrir les concepts de classes et relations. Dans le chapitre suivant, nous aborderons les spécificités des modèles logiques à partir des résultats obtenus dans l'analyse des modèles conceptuels.

3

Étape 3 - Réaliser les modèles logiques

On passe d'un modèle conceptuel à un modèle logique quand on décide de la manière dont on va diviser l'organisation des données en un modèle ou plusieurs. Dans le cas de XML, on tient alors compte des éléments de structuration propres à XML pour établir :

- le découpage du modèle conceptuel en modèles logiques ;
- les frontières entre chacun des modèles logiques ;
- les espaces de noms utilisés ;
- la manière dont seront mis en œuvre les mécanismes de dérivation, d'abstraction et d'unicité ;
- la gestion des attributs ;
- la gestion des associations.

Pour des raisons pratiques, nous n'allons considérer dans ce chapitre que le seul langage d'écriture de schémas XML Schema du W3C. Toutefois, il y a d'autres langages de modélisation de données XML, tels que les DTD, Relax NG, Schematron, et enfin la notation BNF.

Découpage du modèle conceptuel en modèles logiques

Le modèle conceptuel représente le plus souvent une vue globale, pour laquelle les considérations sur l'organisation des données en modules n'ont pas encore été arrêtées. Le découpage en modules de données revêt une importance majeure tant sur le plan de l'architecture d'un système d'information que sur celui de la technique de gestion des données. C'est l'étude de ce découpage qui fait justement l'objet de la mise au point du modèle logique.

VOCABULAIRE Module de données

Nous allons devoir utiliser dans ce chapitre le concept de module de données. C'est un concept important qui fait, à lui tout seul, l'objet des chapitres 9 et 10.
Contentons-nous ici de retenir que toutes les données qui sont utilisées par une application ne peuvent pas être regroupées pêle-mêle dans un même tas. Elles doivent être organisées, structurées et compartimentées afin que les développeurs et tous les intervenants de la programmation s'y retrouvent.
En particulier, le compartimentage de cet ensemble de données en sous-ensembles logiques donne naissance à ce que nous appelons modules de données, à savoir des îlots qui, pour des raisons logiques que nous expliquons dans ce chapitre, doivent se trouver réunis physiquement.

Pour disposer d'un système d'information agile, ce dernier doit être organisé en différentes zones autonomes, reliées par des voies de communication, ainsi que nous l'avons exposé au chapitre 1. Ce découpage en zones autonomes est guidé par des critères métier liés, d'une part, aux fonctions attendues du système et à la manière dont elles sont sollicitées par les processus de l'entreprise, et d'autre part, à l'identification des grands objets de gestion tels que les clients, les fournisseurs, les produits, etc.

Une approche centralisatrice a longtemps fait miroiter l'idée d'un modèle de données universel où les différents points de vue pourraient être réconciliés au sein d'une même base de données. L'évolution incessante des systèmes et des besoins des entreprises a montré qu'il s'agissait d'un rêve : le modèle de données est-il bien le même dans les cadres des processus de vente et du traitement des réclamations ? Ce qui peut apparaître accessoire dans le premier cas sera peut-être essentiel dans l'autre. Il faut donc identifier parmi les données celles qui doivent faire l'objet de blocs indissociables, d'autres qui doivent avoir leur propre cycle de vie et être autonomes.

Sur le plan technique, la modularisation des données permet d'identifier les points d'articulation et de liaison qu'il faudra mettre en place entre chacun des modules de données. En XML, ces points d'articulation se feront à partir d'éléments spécifiques (voir les chapitres 9 sur les éléments purement structurels et 10 sur les modules d'information) et de choix dans les techniques de gestion des liens (voir chapitre 12 sur les liens : IDREF, XLink, etc.). La conception du modèle logique est donc une étape importante dans le cadrage des choix d'architecture de données.

Le point d'équilibre dans la définition des modules se situe entre deux extrêmes : l'un serait l'éclatement maximal du modèle conceptuel en une myriade de petits modèles XML et l'autre le regroupement monolithique de toutes les classes en un seul modèle XML insécable. Atteindre à l'un ou l'autre modèle ne serait que le reflet d'une mauvaise analyse des fonctions et zones du système. Le fractionnement extrême correspond au cas où chaque acteur du système aurait fait valoir son propre point de vue de manière indépendante et où chaque classe du modèle conceptuel donnerait lieu à un module d'information autonome. Cela permet un fort taux de réutilisation des modules mais au prix d'une gestion trop complexe des liens inter-modules. Le regroupement extrême correspond au cas où un acteur dominant aurait réussi à imposer son point de vue aux autres. Il en résulte une organisation monolithique du système, répondant aux seuls besoins de l'acteur dominant.

À travers le cas simplifié de PiloteWeb, nous allons décrire les principaux critères utilisés pour segmenter des modèles conceptuels en modèles logiques :

- identifier les classes majeures et leur périmètre ;
- regrouper les classes dans des modèles distincts ;
- créer les dépendances entre modèles ;
- choisir les espaces de noms.

Identifier les objets majeurs et leur périmètre

Dans la phase d'analyse du modèle conceptuel, nous avons déjà identifié une liste des principaux concepts de PiloteWeb : `pilote`, `site`, `aérodrome`, `restaurant`, `partenaires`, `loueur`. Il nous faut maintenant les examiner un par un pour déterminer leur contour à partir des questions suivantes :

- Quelles associations font partie du domaine de définition de chaque classe ?
- Quelles relations de généralisation/sous-typage font partie du domaine de définition de chaque classe ?
- Quelles classes principales doivent être gérées en commun ?

Analyse des associations

Dans le cas de PiloteWeb, nous commencerons par l'analyse de la classe `Site`. La première tâche à réaliser est de déterminer son contour. À cet effet, une première indication est fournie par les caractéristiques d'association que nous avons définies dans le modèle conceptuel, et, en particulier, les caractéristiques d'agrégation et de composition.

Comme nous l'avons vu au chapitre 2, la composition implique une association forte entre deux classes. La classe `Site` joue le rôle d'agrégat fort dans l'association avec la

classe `Photo`. Cela veut dire que `Photo` est subordonnée à `Site`. Ces deux classes et l'association devront donc être regroupées dans le même modèle logique. Pour toutes les autres associations auxquelles elle participe, la classe `Site` joue uniquement des rôles de subordonnée (en d'autres termes *de partie*). Ces associations et leurs participants sont donc exclus du modèle logique de la classe `site`.

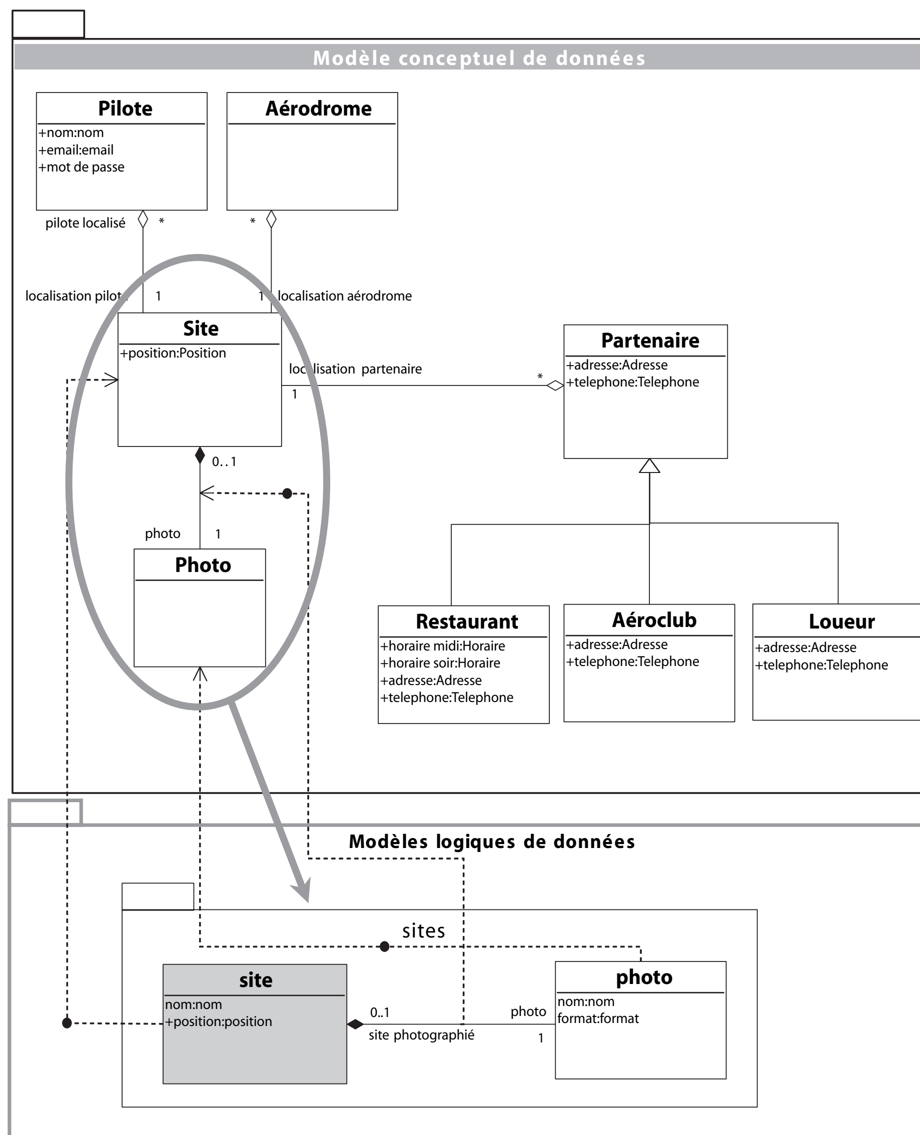


Figure 3–1 Contour de la classe Site

Le modèle logique de la classe `site` est décrit par un nouveau paquetage UML qu'on baptise `sites`, dans lequel on place une copie des classes `Site` et `Photo` du modèle conceptuel (figure 3-1).

> **VOCABULAIRE Paquetage**
>
> Un paquetage UML est le regroupement d'objets du modèle. Les objets regroupés par un paquetage sont les classes, les associations et les relations de généralisation/spécification.

Des relations peuvent être établies entre les classes du modèle conceptuel et leur clone du modèle logique afin de ne pas en perdre la trace. Pour cette raison, ces relations sont dénommées « relations de traçabilité ». Ce tracé s'effectue en utilisant le mécanisme de « dépendance » de UML 2.0. Nous l'avons représenté à la figure 3-2 au moyen de traits en pointillés reliant les classes du modèle logique à leur source dans le modèle conceptuel.

> **VOCABULAIRE Dépendance**
>
> Une dépendance est une relation signifiant qu'un élément de modélisation requiert un autre élément de modélisation pour compléter sa spécification ou son implémentation.

Les modèles logiques présentent le plus souvent des variations importantes avec les modèles conceptuels. C'est pourquoi leurs classes sont obtenues par duplication de celles des modèles conceptuels. Les relations de dépendance permettent de garder la trace de la classe du modèle conceptuel dont provient chaque classe du modèle logique. Il est fait de même pour les associations. Par exemple, dans la figure 3-1, nous indiquons que l'association qui relie les classes `site` et `photo` du modèle logique est une copie de celle qui unit leurs homologues dans le modèle conceptuel.

> **REMARQUE Règle de dénomination des classes des modèles conceptuels et logiques de PiloteWeb**
>
> Afin de distinguer les classes du modèle conceptuel de celles du modèle logique, nous avons adopté dans cet ouvrage la convention suivante : le nom des classes du modèle conceptuel commence par une majuscule tandis que celui des classes du modèle logique commence par une minuscule.

Dans le nouveau paquetage créé, la classe `site` est considérée comme une classe entité, dont les occurrences seront gérées dans le module de données `sites`. Par convention, les classes entités du paquetage sont représentées par un rectangle gris, comme le montre la figure 3-1.

La classe photo, en revanche, n'est pas considérée comme une classe entité du module sites.

REMARQUE Classe entité et UML

UML spécifie une catégorie particulière de classe entité au moyen du stéréotype « entité ». Cependant, cette étiquette apposée à une classe ne permet pas d'en déterminer le contour sous forme d'un module de données. Ce n'est qu'une étiquette.
La solution proposée dans cet ouvrage consiste à définir un contour à une classe entité à l'aide d'un paquetage. Ce paquetage détermine les associations et classes, et seulement celles qui doivent être gérées en commun avec la classe entité.
UML ne propose pas en standard de moyen de désigner la ou les classe(s) entité(s) d'un paquetage. La solution retenue est d'utiliser une relation de dépendance entre paquetage et classe.

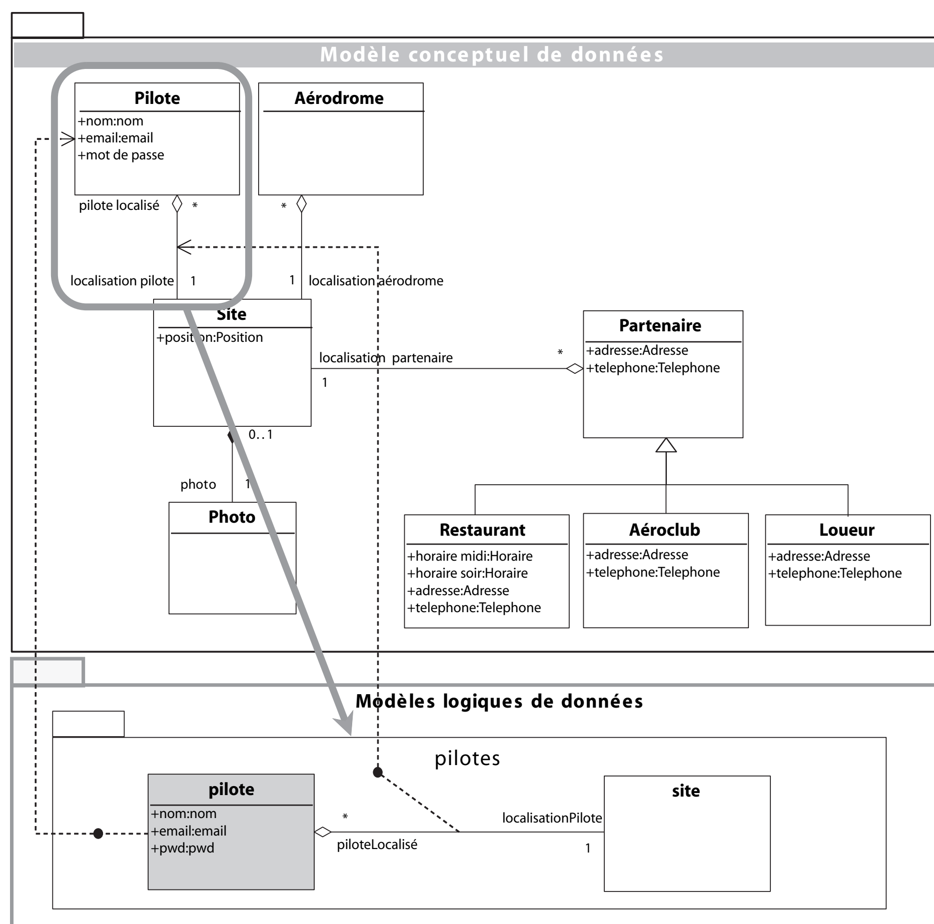


Figure 3–2 Contour de la classe Pilote

Pour la classe `Pilote` du modèle conceptuel, on procède de la même manière que pour la classe `Site` : on crée un nouveau paquetage UML `pilotes` (figure 3-2) et on recherche les classes et associations qui doivent en faire partie à la lecture du modèle conceptuel. La classe `Pilote` a une association avec `Site` (elle sert à indiquer la position géographique du pilote). La classe `Pilote` y joue le rôle d'agrégat, c'est pourquoi l'association doit faire partie du paquetage `pilotes`.

L'inclusion de la classe `site` dans le paquetage `pilotes` montre qu'il peut exister des relations de dépendance entre les paquetages. Sur la figure 3-3, on voit de quelle manière nous pouvons le représenter graphiquement. L'identification de ces dépendances est un point d'analyse important. Celles que nous mettons ici en évidence se traduiront plus tard par des liens entre les modules de données de l'application. On les verra donc réapparaître lorsqu'on va construire les modèles physiques de l'application.

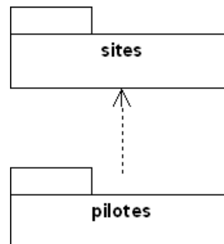


Figure 3–3
Dépendance entre paquetages

Analyse des relations de généralisation/spécialisation

L'analyse de la classe `Partenaire` du modèle conceptuel pose la question du traitement des relations de généralisation/spécialisation, aussi dénommées relations de sur-type/sous-type.

> VOCABULAIRE **Relation de généralisation/spécialisation**
>
> Une relation de généralisation est le résultat d'une opération où les caractéristiques communes à plusieurs classes sont regroupées dans une classe plus générale, aussi appelée sur-type. Réciproquement, une relation de spécialisation est le résultat d'une opération d'enrichissement où des caractéristiques distinctives sont ajoutées aux classes spécialisées, aussi appelées sous-types, en sus des caractéristiques de la classe générale. Les relations de généralisation et de spécialisation vont de pair et sont la réciproque l'une de l'autre.
> De manière plus concise, les experts en modélisation utilisent souvent le terme de « généralisation » pour désigner à la fois la « relation de généralisation » et la « relation de spécialisation ».

Dans cet ouvrage, nous utilisons les seuls termes de sur-type pour la classe la plus générale, de sous-type pour la classe spécialisée et de généralisation pour la relation qui les unit. La figure 3-4 fixe visuellement cette terminologie.

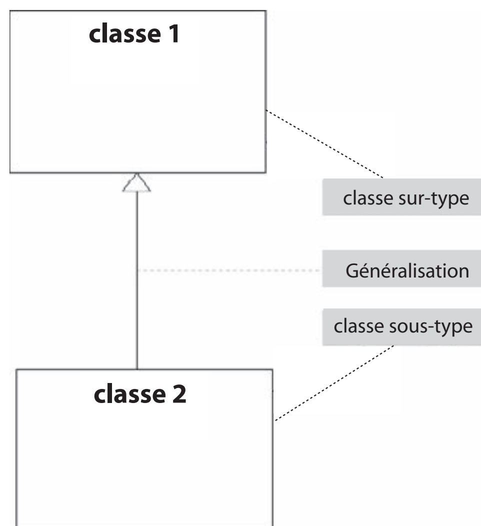


Figure 3–4
Généralisation, sur-type et sous-type

Nous avons expliqué au chapitre 2 que la généralisation correspond à une opération de factorisation par laquelle on réunit sous une même classe les caractéristiques communes à plusieurs autres classes. C'est une opération qui porte sur deux points :

- La mise en commun des caractéristiques des classes (leurs attributs et rôles) ; par exemple, les attributs `Téléphone` et `Adresse` sont réunis dans la classe générale `Partenaire` (figure 3-1), car ils sont communs aux classes `Restaurant`, `Aéroclub` et `Loueur`.
- La mise en commun des occurrences des classes. Ainsi, dans le modèle conceptuel de PiloteWeb de la figure 3-1, les généralisations indiquent que les occurrences des classes `Restaurant`, `Loueur` et `Aéroclub` sont aussi indirectement des occurrences de la classe `Partenaire`.

VOCABULAIRE Occurrence de classe

On appelle occurrence d'une classe sa transposition dans le monde réel. Par exemple, les occurrences des classes UML dont nous parlons seront des instances de « classes Java » ou des éléments XML.
Pour simplifier, et puisque nous sommes dans le monde de la donnée XML, on acceptera l'idée générale que les occurrences des classes que nous présentons seront les éléments XML et leur contenu qui, très concrètement, formeront les documents XML de notre l'application.
Pour désigner l'opération de création d'occurrences, on utilise aussi les termes d'« instanciation » ou d'instance à partir d'une classe. Les termes d'instances et d'occurrences sont donc équivalents.
« Instancier – En programmation orientée objet, créer à partir d'une classe une occurrence de cette classe, héritant par défaut des attributs de sa classe, et qui peut être dotée d'attributs spécifiques » (référence : le Grand Dictionnaire terminologique — Office de la langue française du Québec — `http://www.granddictionnaire.com`).

Dans le cadre de la création des modèles logiques, il faut cependant décider dans quel module de données les occurrences seront gérées physiquement. Si les occurrences de restaurants sont gérées à la fois dans un module `restaurants` et dans un module `partenaires`, on ne saura pas, lors de la pose des liens, quel module cibler pour trouver les restaurants. En effet, pour qu'une occurrence soit adressable, elle doit être localisée dans un et un seul module de données : il faut que le lien soit « biunivoque », c'est-à-dire sans ambiguïté. Dans l'analyse des généralisations que nous conduisons à ce stade de l'élaboration du modèle logique, notre objectif est clairement de déterminer où se trouveront les occurrences physiques des classes qui apparaissent dans les paquetages du modèle logique. À cette fin, nous allons devoir prendre en compte les concepts d'abstraction.

> **VOCABULAIRE Classe abstraite**
>
> Une classe abstraite ne peut pas avoir d'occurrence. Une classe est déclarée abstraite soit parce que sa spécification est incomplète, soit par pure décision de conception.
> Par exemple, dans une hiérarchie de classes comprenant la classe `personne` et ses sous-types `client` et `employé`, il peut être décidé arbitrairement que la classe `personne` est abstraite. Cela signifie que la classe `personne` n'apparaîtra jamais directement dans le système réel mais toujours sous la forme des classes `client` et `employé`. En d'autres termes, le système réel ne gérera pas d'occurrence de `personne` mais uniquement des occurrences de `client` et `employé`.

> **DÉFINITION Classe concrète**
>
> Une classe concrète est une classe qui peut avoir des occurrences.

On distingue trois méthodes de prise en compte des relations de sur-type/sous-type :

- Le sur-type est concret et les sous-types sont abstraits.
- Le sur-type est abstrait et les sous-types sont concrets.
- Le sur-type est concret et les sous-types sont concrets.

Il existe théoriquement un quatrième cas, celui où le sur-type et les sous-types sont abstraits. Dans ce cas, le modèle ne peut pas être implémenté, car il ne peut exister aucune occurrence des classes concernées (en l'occurrence `partenaire`, `restaurants`, `loueurs` et `aroclub`). Ce quatrième cas n'est donc pas retenu.

1er cas. Le sur-type est une classe concrète et les sous-types sont des classes abstraites

Dans ce premier cas, la classe `Partenaire` du modèle conceptuel donne lieu, par duplication, à une classe concrète `partenaire` dans le modèle logique. Les classes `Restaurant`, `Loueur`, `Aéroclub` du modèle conceptuel donnent lieu, par duplication, aux classes abstraites `restaurant`, `loueur`, `aeroclub` dans le modèle logique.

Les occurrences de la classe `partenaire` ont au choix les caractéristiques des classes abstraites `restaurant`, `loueur` ou `aéroclub` ; il n'existe pas, à proprement parler, d'occurrences de restaurants, loueurs ou aéroclubs.

Plusieurs solutions d'implémentation se présentent : un typage statique ou dynamique de la classe `partenaire`.

Pour mettre en œuvre le typage statique, on opère de la manière suivante :

- On ajoute un attribut `typepartenaire` à la classe `partenaire` du modèle logique. Cet attribut se traduira également dans le XML par un attribut de même nom (voir tableau 3-1). Il a pour type une énumération dont les valeurs sont `restaurant`, `loueur`, `aeroclub`.
- On inclut dans la classe `partenaire` du modèle logique l'ensemble des attributs et associations des classes `restaurant`, `loueur` et `aeroclub`. Dans le modèle XML correspondant, on regroupe dans un même élément toutes les caractéristiques des restaurants, aéroclubs et loueurs. Il reviendra donc à l'application de savoir appliquer les bons traitements ou produire les bons sous-éléments en fonction de la valeur de l'attribut `typepartenaire`. Cette solution est celle retenue le plus souvent dans les applications de XML au monde relationnel qui ne dispose pas de possibilité de typage dynamique.

Ce cas de figure est illustré au tableau 3-1.

Tableau 3–1 Généralisation : typage statique

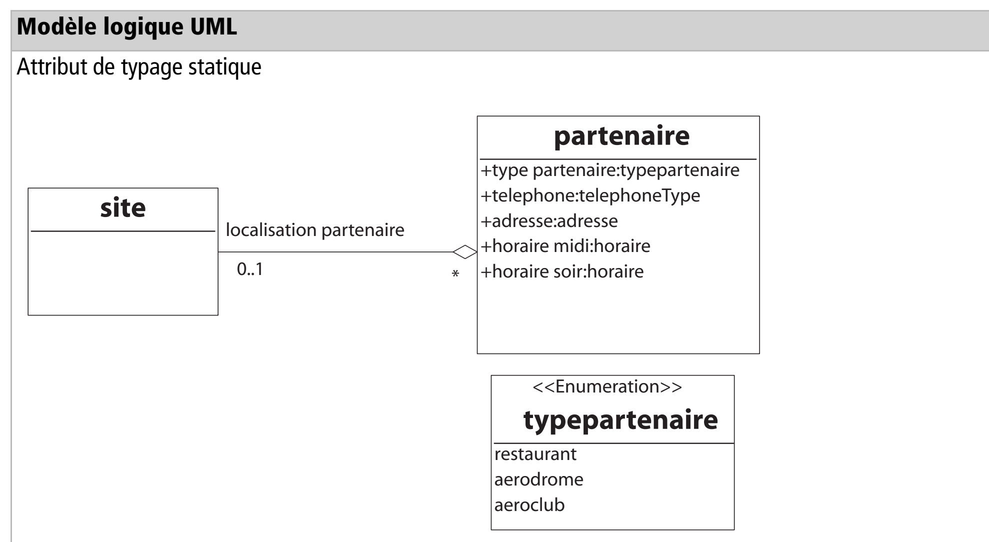


Tableau 3–1 Généralisation : typage statique

Représentations XML possibles et explications
Implémentation sous forme d'un attribut discriminant

```
<xs:element name="partenaire">
  <xs:complexType>
   <xs:sequence>
     <xs:element name="localisationPartenaire"
                 type="entityRef"/>
     <xs:element name="adresse" type="adresse"/>
     <xs:element name="telephone"
                 type="telephone"/>
     <xs:element name="midi" type="horaire"/>
     <xs:element name="soir" type="horaire"/>
   </xs:sequence>
   <xs:attribute name="typepartenaire"
                 use="required">
     <xs:simpleType>
       <xs:restriction base="xs:string">
         <xs:enumeration value="restaurant"/>
         <xs:enumeration value="loueur"/>
         <xs:enumeration value="aeroclub"/>
       </xs:restriction>
     </xs:simpleType>
   </xs:attribute>
 </xs:complexType>
</xs:element>
```

Pour mettre en œuvre le typage dynamique, on opère de la façon suivante :

- La classe `Partenaire` du modèle conceptuel donne lieu à une classe abstraite `partenaireType` et une classe concrète `partenaire`.
- Les sous-types de la classe `Partenaire` dans le modèle conceptuel – `Restaurant`, `Loueur`, `Aéroclub` – donnent lieu à des classes abstraites dans le modèle logique : `restaurantType`, `loueurType` et `aeroclubType`. Ces classes sont des sous-types de la classe abstraite `partenaireType`.

Dans le schéma XML correspondant, la classe `partenaire` devient l'élément `partenaire` dont le type, abstrait, est `partenaireType`. Trois types sont définis qui sont `restaurantType`, `loueurType` et `aeroclubType`. Ils peuvent se substituer au type abstrait car ils en sont des dérivés valides. Aussi, dans les documents XML conformes à ce schéma, il sera obligatoire d'utiliser l'attribut `xsi:type` sur chaque élément partenaire pour indiquer son type concret.

Tableau 3–2 Réalisation d'un typage dynamique

Modèle logique UML

Classe concrète avec un type abstrait

partenaireType
{Abstract}
+Adresse:adresse
+telephone:telephone

loueurType
{Abstract}
+nom:nom

aeroclubType
{Abstract}
+nom:nom

restaurantType
{Abstract}
+horaire midi:horaire
+horaire soir:horaire
+nom:nom

Partenaire

Représentations XML

Implémentation sous forme de type dynamique

```
<xs:complexType name="partenaireType" abstract="true">
  <xs:sequence>
    <xs:element name="localisationPartenaire"
                type="entityRef"/>
    <xs:element name="adresse" type="adresse"/>
    <xs:element name="telephone" type="telephone"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="restaurantType">
  <xs:complexContent>
    <xs:extension base="partenaireType">
      <xs:sequence>
        <xs:element name="midi" type="horaire"/>
        <xs:element name="soir" type="horaire"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="partenaire" type="partenaireType">
```

Tableau 3–2 Réalisation d'un typage dynamique

Exemple de document instance

```
<partenaire xsi:type="restaurantType">
  <pw:localisationRestaurant xl:type="simple"
                             xl:href="http://www.pilotweb.com#"/>
  <pw:adresse>String</pw:adresse>
  <telephone>0230548967</telephone>
  <midi>String</midi>
  <soir>String</soir>
</partenaire>
```

2e cas. Le sur-type est une classe abstraite et les sous-types sont des classes concrètes

La classe `Partenaire` du modèle conceptuel donne lieu à une classe abstraite `partenaireType` dans le modèle logique. Elle est simplement utilisée pour factoriser des caractéristiques propres aux partenaires. Il n'y a pas d'occurrence de partenaire mais seulement des occurrences de restaurant, d'aéroclub et de loueur. Chaque sous-type de la classe `Partenaire` dans le modèle conceptuel donne donc lieu à une classe concrète dans le modèle logique : `restaurant`, `loueur`, `aeroclub`. Ces classes héritent des caractéristiques de la classe abstraite `partenaireType`.

La correspondance avec le schéma XML répond aux règles suivantes :

- La classe abstraite `partenaireType` donne lieu à un type complexe du même nom.
- Les classes concrètes `restaurant`, `loueur`, `aeroclub` donnent chacune lieu à un élément dont le type est une extension de `partenaireType`.

Ce cas de figure est illustré au tableau 3-3.

Tableau 3–3 Le sur-type est une classe abstraite

Modèle logique UML

Classe concrète avec un type abstrait

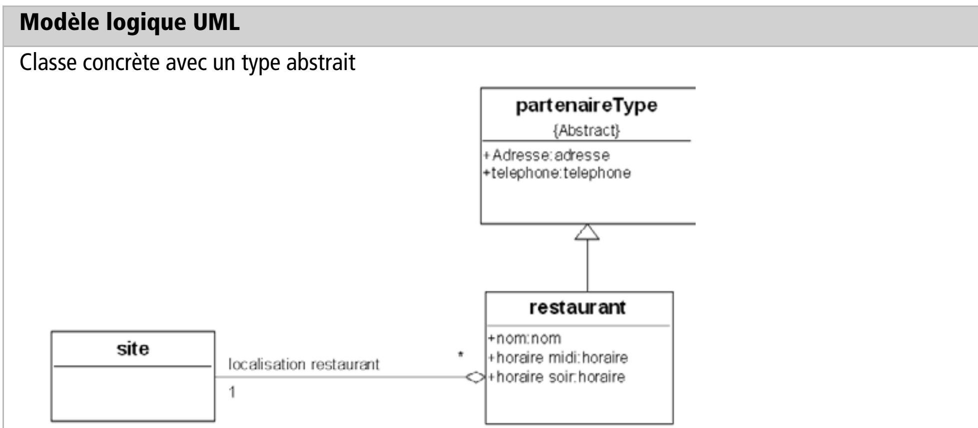


Tableau 3–3 Le sur-type est une classe abstraite

Représentations XML possibles et explications
La classe concrète donne lieu à un élément global définissant un type local basé sur un type abstrait

```
<xs:complexType name="partenaireType" abstract="true">
  <xs:sequence>
    <xs:element name="adresse" type="adresse"/>
    <xs:element name="telephone" type="telephone"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="restaurant">
   <xs:complexType>
    <xs:complexContent>
      <xs:extension base="partenaireType">
        <xs:sequence>
          <xs:element
              ref="localisationRestaurant"/>
          <xs:element name="midi" type="horaire"/>
          <xs:element name="soir" type="horaire"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

3e cas. Le sur-type est une classe concrète et les sous-types sont des classes concrètes

Ce cas correspond à la transformation des généralisations du modèle conceptuel en relations associatives dans le modèle logique selon les règles suivantes :

- Chaque classe participant à la généralisation dans le modèle conceptuel donne lieu à une classe concrète dans le modèle logique. Dans PiloteWeb, les classes `Partenaire`, `Restaurant`, `Loueur`, `Aéroclub` du modèle conceptuel donnent lieu aux classes concrètes `partenaire`, `restaurant`, `loueur`, `aéroclub` du modèle logique.
- Chaque généralisation du modèle conceptuel est transformée dans le modèle logique en une association d'agrégation. Dans le cas de PiloteWeb (figure 3-5), cela amène la classe `partenaire` à avoir trois nouvelles associations : une pour chacune des classes `restaurant`, `loueur` et `aeroclub`.
- Pour chacune de ces associations, la classe issue du sur-type joue le rôle de subordonnée, et celle issue du sous-type celui d'agrégat. Dans PiloteWeb, la classe `partenaire` est la classe subordonnée.
- La multiplicité du rôle subordonné est égale à `1` et celle du rôle *agrégat* est égale à `*`.

- Le nom du rôle subordonné doit être représentatif des données des sous-types mises en commun par la généralisation. Dans le cas de PiloteWeb, la classe `partenaire` joue le rôle `fichePartenaire` parce qu'on a jugé que ce nom représentait bien l'ensemble des données qu'elle généralise.

La transformation des sur-types et sous-types en classes concrètes est la solution retenue pour la suite de notre exemple PiloteWeb. Elle simplifie la représentation en XML du modèle logique en évitant de recourir aux techniques d'extension ou de restriction de type de XML Schema.

Cette solution est présentée à la figure 3-5.

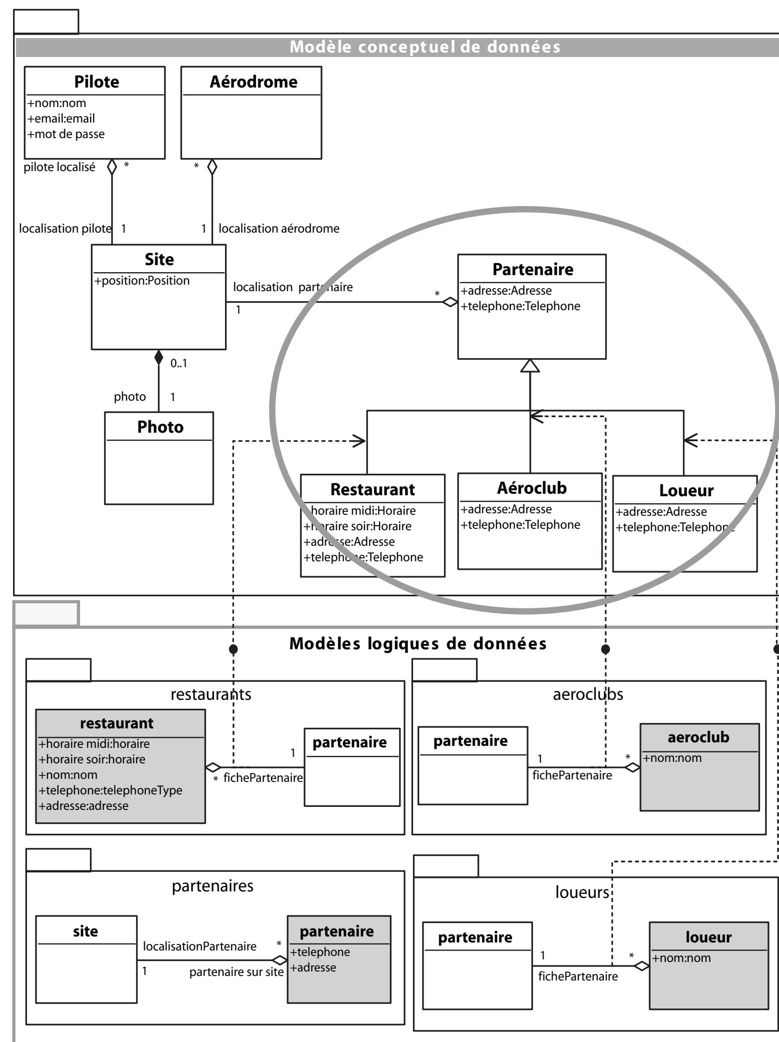


Figure 3–5
Transformation des généralisations en associations

Modèles et espace de noms

La première phase d'analyse des modèles logiques a permis d'identifier les modules de données à l'aide de paquetages UML (sites, pilotes, restaurants, aéroclubs, etc., des figures 3-1, 3-2 et 3-5) et de leurs dépendances les uns par rapport aux autres. À la figure 3-6, vous verrez comment nous les regroupons en un paquetage unique représentant finalement le modèle logique global de PiloteWeb.

C'est ce paquetage final qui porte l'ensemble du vocabulaire de PiloteWeb. Il joue donc le rôle d'espace de noms. Aussi perçoit-on les deux fonctions du concept de paquetage d'UML :

- L'une consiste à définir les frontières des modules de données : il s'agit des paquetages individuels, sites, pilotes, restaurants, etc.
- L'autre consiste à définir l'espace de noms de l'application : il s'agit du paquetage baptisé `PiloteWeb::Modèles logiques de données` de la figure 3-6. À ce paquetage est attribué un URI qui servira d'espace de noms XML, par exemple : http://www.piloteweb.com/2005.

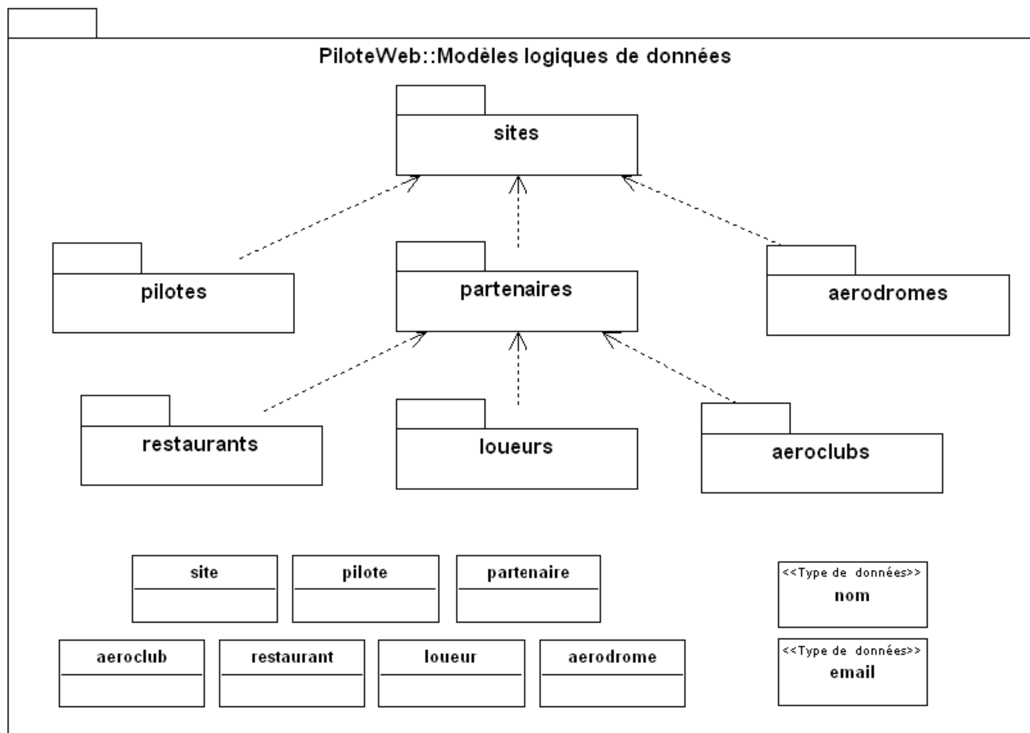


Figure 3–6 Paquetages de PiloteWeb

VOCABULAIRE Objets de modélisation

Un objet de modélisation est pour UML tout constituant d'un paquetage. Plus précisément, UML utilise la terminologie d'« élément de modélisation ». Or, nous avons considéré que cette terminologie pouvait prêter à confusion avec la notion d'élément de XML. Aussi avons-nous choisi d'utiliser le mot objet à la place du mot élément.

Pour indiquer comment chacune de ces deux fonctions est mise en œuvre dans le modèle UML, il faut que l'on précise les notions d'objets de modélisation, détenus et importés. UML fait la distinction entre les objets de modélisation détenus par un paquetage et ceux importés par un paquetage. Seuls les noms des éléments détenus font partie de l'espace de noms défini par le paquetage. Les éléments importés proviennent d'autres paquetages dont ils conservent l'espace de noms.

VOCABULAIRE Objets de modélisation détenus par un paquetage

On dit d'un objet de modélisation qu'il est détenu par un paquetage quand son nom appartient à l'espace de noms défini par le paquetage. Un objet de modélisation ne peut être détenu que par un et un seul paquetage. Lorsque le paquetage est supprimé, les objets de modélisation qu'il détient le sont aussi. Dans le cadre de la modélisation de données, les objets de modélisation qui peuvent être détenus par un paquetage sont les classes, les associations et les généralisations.

VOCABULAIRE Objets importés par un paquetage

On dit d'un objet de modélisation qu'il est importé dans un paquetage quand son nom est détenu par un paquetage différent de celui qui l'importe. On dit aussi de ces objets qu'ils sont référencés. Lorsque le paquetage qui fait l'import est supprimé, les objets importés ne sont pas pour autant supprimés, seules leurs références disparaissent.
Dans le cadre de la modélisation de données, les objets de modélisation qui peuvent être importés par un paquetage sont les classes, les associations et les généralisations.

Les notions de détention et d'importation permettent de régler la gestion des espaces de noms à partir du modèle logique. Une bonne règle de modélisation consiste à séparer les paquetages qui jouent le rôle d'espaces de noms de ceux qui jouent le rôle de modèles de données. Les paquetages « espaces de noms » sont les détenteurs des classes, associations, généralisations, ainsi que des paquetages décrivant les modèles de données. Les paquetages de type « modèles de données » n'utilisent que le mécanisme d'import pour déterminer leur contenu.

Dans PiloteWeb, le paquetage logique global fait office d'espace de noms. Il détient en conséquence toutes les classes et associations, comme cela apparaît à la figure 3-6. Les sous-paquetages sites, pilotes, partenaires, etc., ne détiennent pas de classes ou associations et, à ce titre, ne sont pas des espaces de noms. Ils indiquent les classes et associations qui font partie de leur périmètre.

Le mécanisme d'importation est malheureusement souvent laissé pour compte. Les concepteurs UML privilégient généralement le mécanisme de détention pour ranger les classes dans les paquetages ; ils ignorent le mécanisme d'import. Ainsi la classe `pilote` appartient-elle au paquetage `pilotes`. Le défaut de cette approche est de multiplier les espaces de noms puisque dans UML tout paquetage détenteur est en même temps espace de noms pour les objets de modélisation qu'il détient. En suivant cette approche, nous aurions des espaces de noms pour les sites, pilotes, aérodromes, etc., ce qui rendrait la correspondance avec XML pour le moins complexe car il n'est pas recommandé, avec XML Schema, de multiplier le nombre des espaces de noms d'une application.

Dans une bonne analyse, l'espace de noms définit le champ sémantique du domaine étudié, c'est-à-dire les frontières de vocabulaire. Dans le cas PiloteWeb, on ne désire pas créer des vocabulaires distincts, l'un pour les pilotes, l'autre pour les aérodromes, etc. Dans le périmètre de notre étude, on se borne à établir le vocabulaire de PiloteWeb.

Dans le passage à XML, seul le paquetage `PiloteWeb::Modèles logiques de données` donne lieu à un espace de noms. En fonction des choix d'implémentation physique, on pourra disposer d'un seul fichier schéma regroupant tous les sous-paquetages, ou d'autant de fichiers schémas qu'il y a de sous-paquetages : `sites.xsd`, `pilotes.xsd`, etc.

Toutes les classes détenues par le paquetage global vont donner lieu à des éléments globaux dans le schéma XML. Cependant, certaines classes n'ont pas vocation à devenir des éléments globaux parce qu'elles n'ont qu'un emploi localisé. C'est par exemple le cas de la classe `photo`, qui n'intervient que dans le cadre de la définition de la classe `site`. De même, certains types de données n'ont de cas d'emploi que pour un seul attribut d'une classe, par exemple le type de données `format`, qui n'est utilisé que pour l'attribut `format` de la classe `photo`. Dans ces cas d'utilisation locaux d'une classe ou d'un type de données, il faut pouvoir changer leur détenteur. Au lieu de les faire détenir par un paquetage, il est envisageable de les faire détenir par une classe. Cette classe détentrice joue alors elle-même un rôle d'espace de noms. Ainsi la classe `photo` est-elle détenue par la classe `site` et le type de données `format` est lui-même détenu par la classe `photo`. Il en découle la règle suivante :

- Les classes et types de données utilisés localement par rapport à une classe doivent être détenus localement par cette classe.
- Le nom de la classe détentrice joue alors le rôle d'espace de noms local pour les classes et types de données ainsi détenus.

La notation UML représente ces relations de détention par une ligne avec un ornement du côté du détenteur en forme de cercle barré d'une croix (figure 61 dans la spécification UML 2.0).

Tableau 3–4 Détention locale de classes

Modèle logique UML

Classes locales

Représentations XML

Types locaux

```
<xs:element name="site">
  <xs:complexType>
    <xs:attribute name="nom"/>
    <xs:sequence>
      <xs:element name="Photo">
        <xs:complexType>
          <xs:attribute name="nom"/>
          <xs:attribute name="format">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="png"/>
                <xs:enumeration value="gif"/>
                <xs:enumeration value="jpg"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Classe, type XML et élément XML

Dans cette section, nous allons préciser les correspondances existant entre les classes UML et les éléments et types XML. L'un des points dont nous allons particulièrement tenir compte est la double fonction des classes UML :

- Donner les caractéristiques d'un objet. Ainsi, la classe `site` a comme caractéristiques un nom, des positions géographiques (x,y) et une description visuelle qui prend la forme d'une photo.
- Donner une existence à une liste d'objets. Ainsi, le modèle logique de PiloteWeb où la classe `site` est une classe concrète (figure 3-1) indique qu'il existe des occurrences précises de site, par exemple : `le Mans-Arnage, position(x=47.9486, y=-0.2016), photo (nom="lemans.jpg" format="jpg")` ; `Toussus-le Noble, position(x=48.7494,y=-2.1108), photo(nom="toussus.jpg",format="jpg")` ; etc.

Nous avons vu avec la classe `partenaire` que les généralisations peuvent introduire des ambiguïtés sur le mode d'existence des objets : un partenaire existe-t-il sous la forme d'un partenaire, d'un restaurant, d'un aérodrome ou d'un aéroclub ? Nous avons vu comment lever ces ambiguïtés en indiquant les classes qui sont abstraites et celles qui sont concrètes.

De son côté, XML Schema fait une distinction explicite entre la définition des caractéristiques et la déclaration des listes possibles d'occurrences. La définition des caractéristiques est faite au moyen des types, celle des occurrences au moyen des éléments. On dit ainsi qu'il peut y avoir des éléments `site` de type `siteType`. Le type `siteType` est soit explicite donc global, soit implicite donc local (auquel cas, il n'a pas de nom).

À partir des explications précédentes, il apparaît que toute classe UML donne lieu à la fois à un type XML et à un élément XML, à l'exception des classes abstraites qui ne donnent lieu qu'à un type global.

- Classe concrète ⇒ élément XML + type XML.
- Classe abstraite ⇒ type XML global.

Il faut maintenant déterminer si l'élément XML correspondant à une classe concrète est global ou local. Nous avons vu dans la section sur les espaces de noms que toute classe détenue par le paquetage ayant la fonction d'espace de noms donnait lieu à un élément global. Au contraire, toute classe détenue par une autre classe donne lieu à un élément local.

- Classe détenue par le paquetage d'espace de noms ⇒ élément XML global.
- Classe détenue par une autre classe ⇒ élément XML local.

La règle concernant le caractère global ou local des types XML correspondant aux classes concrètes est plus délicate. Toute classe d'une généralisation se traduit par un type XML global. En effet, les mécanismes de dérivation de XML Schema – les

seuls à pouvoir traduire le concept de généralisation de UML – n'existent que pour des types globaux de XML Schema. Dans le cas de PiloteWeb où la classe `partenaire` est une généralisation des classes `restaurant`, `loueur` et `aeroclub`, on la traduira par le type global `partenaireType` :

```
<xs:complexType name="partenaireType">
  <xs:sequence>
    <xs:element ref="localisationPartenaire"/>
    <xs:element name="adresse" type="adresse"/>
    <xs:element name="telephone" type="telephone"/>
  </xs:sequence>
</xs:complexType>
```

Pour les autres cas, il y a deux écoles : celle des méthodes de transformation automatique et celle cherchant à optimiser le schéma XML. Les outils de transformation des modèles de classes en schémas XML privilégient la création systématique de types globaux. C'est le cas des règles de transformation adoptées à l'OMG pour XMI 2.0. Ces règles sont exprimées en BNF. La règle 3 (tableau 3-5) indique qu'une classe est représentée par une déclaration de type et une déclaration d'élément.

Tableau 3–5 Déclaration de production de schéma dans la spécification XMI de l'OMG

Extrait de la syntaxe EBNF	Description
`3. ClassSchema ::= 4:ClassTypeDef 5:ClassElementDef`	`3. The class schema contribution consists of a type declaration based on the attributes and references of the class, and an element declaration for the Class itself.` Dans un schéma XML, la classe est transposée au moyen d'une définition de type, basée sur les attributs et références de la classe, et d'une définition d'élément pour la classe elle-même.

La seconde école ramène le nombre de types globaux aux seuls types réutilisés par plusieurs éléments. L'objectif est de diminuer le volume des schémas et de limiter les conséquences de l'utilisation d'espaces de noms.

Une fois que l'on a défini les éléments et les types du schéma XML issus directement de l'analyse des classes UML, il faut déterminer s'il y a lieu d'ajouter des éléments XML supplémentaires pour gérer les listes d'éléments : faut-il des éléments `photos`, `sites`, etc. ?

Les éléments pour lesquels cette organisation en liste est nécessaire sont les classes entités des modules de données. Chaque classe entité d'un module de données a vocation à être adressée par d'autres classes appartenant à d'autres modules de données.

Il en découle la règle suivante :

- À chaque classe entité doit correspondre un élément XML gérant la collection des occurrences de cette classe.
- Par convention, le nom de l'élément XML gérant la collection des occurrences est le nom de la classe auquel on ajoute un « s » comme indiqué à la figure 3-6.
- Les modules de données contiendront les occurrences de leurs classes entités. L'analyse physique des modèles de données (voir chapitre 4) permettra de savoir si les modules de données seront gérés dans un même et seul document XML, ou au contraire dans différents documents.

Tableau 3–6 Déclaration de collection d'éléments

Modèle UML
Un module et sa classe entité
sites site nom:nom +position:position
Représentations XML
Module et collection d'éléments

```
<xs:element name="site" type="pw:siteType"/>

<xs:element name="sites">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="pw:site"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Les éléments XML tels que `sites` n'ont donc pas de correspondance directe en tant que classes dans le modèle UML. Ils sont la manifestation d'une collection d'éléments organisés en module de données.

Outre la gestion des collections, il faut permettre l'identification des éléments XML contenus dans ces collections. On peut y procéder soit au travers de caractéristiques propres à la classe, tel un attribut `nom`, soit au moyen d'un mécanisme standard utilisant un code d'identification attribué automatiquement. C'est la solution qui est

retenue dans PiloteWeb. Dans le schéma XML, les éléments `site`, `pilote`, `partenaire`, `restaurant`, `aeroclub` et `aerodrome` ont tous un attribut `id` de type `xs:ID`.

```
<xs:element name="site">
  <xs:complexType>
    <xs:sequence>
      …
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:attribute ref="pw:nom"/>
  </xs:complexType>
</xs:element>
```

Gestion des associations

En étudiant les associations, nous allons compléter nos analyses sur :

- les collections d'éléments XML ;
- la définition du type des éléments XML.

Cependant, pour appréhender ces sujets, revenons à la définition d'une association et aux objets qui la composent :

- Les références d'objets permettent de désigner un objet par un pointeur, une adresse ou tout autre mécanisme de référencement.
- Les significations données à ces références. Par exemple, dans l'association entre la classe `site` et la classe `pilote`, la signification attribuée à la référence qui permet de localiser un pilote au moyen d'une référence à un site est `localisationPilote`.

Pour qu'il y ait association, il faut d'abord pouvoir faire référence à des objets identifiables et adressables. Les associations organisent ensuite les références aux objets en leur attribuant des rôles afin d'en indiquer la signification comme nous l'illustrons à la figure 3-7. L'association précise ainsi l'existence de la liaison entre deux objets et la raison d'être de cette liaison ; c'est ce que l'on appelle doctement la sémantique des liens. Dans PiloteWeb, l'association entre la classe `pilote` et la classe `site` indique qu'il est possible de référencer des sites au titre du rôle `localisationPilote` et qu'il est aussi possible de référencer des pilotes au titre du rôle `piloteLocalisé`. L'association indique en outre que les rôles `localisationPilote` et `piloteLocalisé` vont de pair.

Outre la sémantique des liens, les associations permettent de décrire tous les modes de navigation du lien ; les associations ne sont pas directionnelles, seul l'usage que

l'on en fait l'est. Ainsi peut-on naviguer des pilotes localisés vers les sites de localisation, aussi bien que des sites de localisation vers les pilotes localisés.

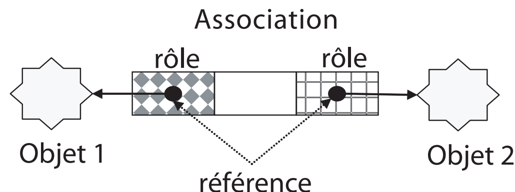


Figure 3–7
Association et références

Cependant, la gestion des références requise pour une mise en œuvre complète des associations est le plus souvent complexe à réaliser car cela demande de prendre en compte tous les modes de traversée des associations. Dans les modèles physiques, cela suppose que l'on dispose d'un mécanisme gérant explicitement les associations. Par exemple, dans un document XML, il faudrait disposer d'un élément spécifique pour chaque association, comme indiqué dans l'exemple simplifié ci-après :

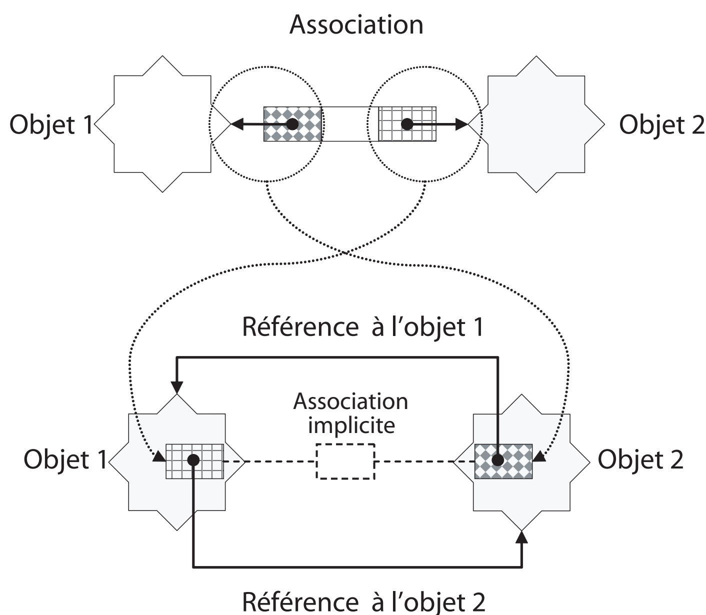
```
<objet1>
  …
</objet1>
<objet2>
  …
</objet2>
<association objet1-objet2>
  <refObjet id=objet1>
  <refObjet id=objet2>…
</association>
```

Pour les applications standards, un tel choix d'implémentation est bien trop coûteux : pour passer de l'objet 1 à l'objet 2, il faut effectuer trois navigations : `objet1 -> association(refObjet id=objet1) -> association(refObjet id=objet2) -> objet2`. Pour simplifier la navigation, l'élément association est supprimé, chacune des références « migrant » alors dans les objets concernés comme indiqué dans l'exemple suivant :

```
<objet1>
  <refObjet id=objet2>…
  …
</objet1>
<objet2>
  <refObjet id=objet1>
  …
</objet2>
```

Le cas général est présenté à la figure 3-8 :

Figure 3–8
Références ayant migré dans les objets



Les règles de transfert des associations dans les objets dépendent de l'analyse du type des associations comme nous allons le voir.

Les associations de composition

Les associations de composition sont, comme nous l'avons dit en début de chapitre, des associations de type tout/partie exprimant une subordination forte entre deux classes comme l'indique la figure 3-9. Le rôle représentant *le tout* est le rôle de composition. Il désigne la classe subordonnante dans l'association. L'autre rôle représente *les parties* ; c'est le rôle de subordination. Il indique la classe subordonnée dans l'association.

Figure 3–9 Association de composition

Dans une composition, les occurrences de la classe subordonnée ne peuvent exister sans l'occurrence de la classe subordonnante. Ainsi, dans PiloteWeb, les occurrences de la classe `photo` n'existent que lorsque celles de la classe `site` existent. Cela indique que les associations de composition doivent se traduire dans les documents XML par des imbrications d'éléments. Il en résulte les règles de transformation suivantes :

- L'association de composition se traduit par la déclaration d'un ou plusieurs élément(s) dans la définition du type de la classe subordonnante de ces éléments. Par exemple, le type représentant la classe `site` contient la déclaration de l'élément `photo` (et non un lien à cet élément) qui est subordonné à cette classe.
- L'élément prend pour nom celui du rôle de subordination. Dans l'exemple précédent, le nom du rôle de subordination est le même que le nom de la classe subordonnée à savoir `photo`. Le type XML de cet élément est celui de la classe subordonnée et pourra être défini localement ou globalement. Dans notre exemple, selon que la classe `photo` a donné lieu à un type global ou local, l'élément `photo` sera défini globalement ou localement (le tableau 3-7 contient des exemples de ces deux cas).
- Comme valeurs de ses indicateurs d'occurrence, on reprend celles données au subordonné : l'attribut `minOccur` (respectivement `maxOccur`) de XML Schema correspond à la multiplicité minimale (respectivement maximale) du subordonné dans un modèle UML. Dans notre exemple, la multiplicité `1` de la classe subordonnée `photo` indique que toute occurrence de la classe `site` contiendra une et une seule occurrence de la classe `photo`. Dans le schéma XML, nous avons donc mis à `1` les multiplicités de l'élément `photo` : `minOccurs="1"` et `maxOccurs="1"`.

En ce qui concerne le sens de parcours de l'association, aussi bien dans le sens subordonnant vers subordonné que l'inverse, XML permet de l'assurer sans aucune difficulté par la simple exploitation des mécanismes de parcours d'arbre XML : element.enfant(x)←→ element.parent.

Tableau 3–7 Association de type composition

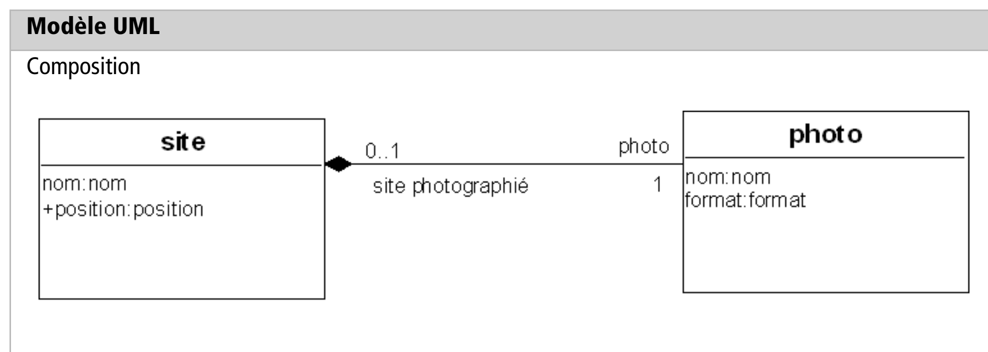


Tableau 3–7 Association de type composition

Représentations XML

Imbrication d'élément avec une déclaration de type local
 	
`<xs:element name="site">`
`  <xs:complexType>`
`    <xs:sequence>`
`      <xs:element name="photo" minOccurs="1" maxOccurs="1">`
`        <xs:complexType>`
`          <xs:attribute name="nom"/>`
`          <xs:attribute name="format">`
`            <xs:simpleType>`
`              <xs:restriction base="xs:string">`
`                <xs:enumeration value="png"/>`
`                <xs:enumeration value="gif"/>`
`                <xs:enumeration value="jpg"/>`
`              </xs:restriction>`
`            </xs:simpleType>`
`          </xs:attribute>`
`        </xs:complexType>`
`      </xs:element>`
`      …`
`    </xs:sequence>`
`    …`
`  </xs:complexType>`
`</xs:element>`

Imbrication d'élément avec une déclaration de type global
 	
`<xs:complexType name="photoType">`
`  ...`
`</xs:complexType>`
 	
`<xs:element name="site">`
`  <xs:complexType>`
`    <xs:attribute name="nom"/>`
`    <xs:sequence>`
`     <xs:element name="photo"`
`                  type="photoType"`
`                  minOccurs="1"`
`                  maxOccurs="1"/>`
`    </xs:sequence>`
`  </xs:complexType>`
`</xs:element>`

Les associations d'agrégation

Introduction

Les associations d'agrégation sont des associations de type tout/partie exprimant une subordination faible entre deux classes comme l'indique la figure 3-10. Le rôle représentant *le tout* est le rôle d'agrégation. Il désigne la classe subordonnante dans l'association. L'autre rôle représente *les parties* ; c'est le rôle de subordination. Il indique la classe subordonnée dans l'association. Dans une agrégation, contrairement à la composition, les occurrences de la classe subordonnée peuvent exister indépendamment de l'occurrence de la classe subordonnante. Ainsi, dans PiloteWeb, les occurrences de la classe `site` peuvent exister indépendamment de celles de la classe `pilote`. Pour autant, les sites font *partie* de la définition d'un pilote.

Figure 3–10 Association d'agrégation

Règles par défaut

Par défaut, la traduction en objets XML des associations d'agrégation obéit aux règles suivantes :

- Chaque rôle de subordination donne lieu à la déclaration d'un élément dans la définition du type XML représentant la classe subordonnante. Par exemple, le type XML représentant la classe `pilote` contient une déclaration d'élément pour représenter l'objet associé `site`.
- Le nom donné à cet élément est celui du rôle qu'il a dans l'association. Dans notre cas, l'élément déclaré ne sera pas `site` mais `localisationPilote`.
- Les valeurs des indicateurs d'occurrence de l'élément XML sont celles de la multiplicité du rôle qui lui correspond : l'attribut `minOccur` (respectivement `maxOccur`) de XML Schema aura la valeur de la multiplicité minimale (respectivement maximale) du modèle de classe UML.
- La caractéristique particulière de l'agrégation est que l'objet associé peut avoir une vie propre indépendamment de l'objet qui l'agrège. Cela signifie qu'il revient à la classe agrégeant de gérer le lien qui l'unit à l'objet subordonné, et pas l'inverse. Ainsi, les agrégations traduisent implicitement une « navigation sémantique »

entre les objets par la seule existence de ce lien directif qui, pour former l'agrégation, va de la classe agrégeant à l'objet agrégé. La figure 3-11 illustre cette caractéristique particulière des agrégations.

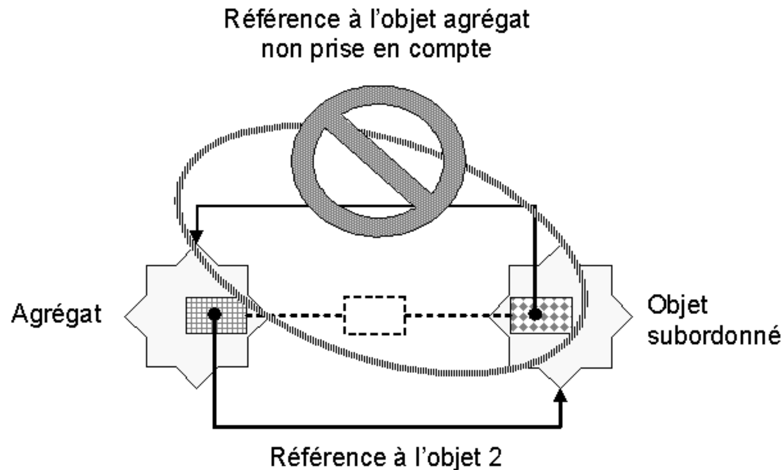


Figure 3–11
Agrégation et navigabilité

XML et XML Schema ne proposant pas intrinsèquement une méthode de représentation des références, ou liens, il est conseillé, au niveau du modèle logique, de créer un type XML permettant de différer le choix de la solution qui sera finalement retenue pour représenter ces références. Dans cette section, nous proposons un type XML appelé `entityRef` basé sur XLink :

```
<xs:complexType name="entityRef">
  <xs:attribute ref="xl:type" fixed="simple"/>
  <xs:attribute ref="xl:href" use="required"/>
</xs:complexType>
```

Dans l'exemple décrit au tableau 3-8, la référence au `site` depuis l'élément `pilote` utilise pour l'élément `localisationPilote` le type `entityRef`. C'est un nom générique qui peut représenter toute sorte de mécanisme de référencement d'entités. Lors de la conception du modèle physique, ce type sera adapté aux options de stockage physique choisies.

En lieu et place d'éléments, on peut utiliser des attributs pour représenter les références. Les règles précédemment édictées deviennent alors :

- Chaque rôle que prend l'objet subordonné par rapport à un subordonnant se traduit par un attribut dans le type XML représentant la classe qui subordonne. Par

exemple, le type XML de la classe `pilote` aura dans ce cas un attribut dont le nom est `localisationPilote` (et non un élément comme dans le cas précédent).

- Le type de cet attribut sera également le type générique `entityRef`.

L'avantage de la gestion des références par attribut, c'est sa simplicité. Ses inconvénients n'en sont pas moins nombreux :

- Il n'est pas possible de contrôler les indicateurs d'occurrence.
- Le contrôle de la structure des références complexes est limité aux possibilités offertes par les types simples de XML.

Tableau 3–8 Association de type agrégation

Modèle UML

Agrégation

pilote
+nom:nom
+email:email
+pwd:pwd
*
localisationPilote
piloteLocalisé
1
site

Représentations XML

Référence au moyen d'un élément XML

```
<xs:complexType name="entityRef">
  <xs:attribute ref="xl:type" fixed="simple"/>
  <xs:attribute ref="xl:href" use="required"/>
</xs:complexType>

<xs:element name="pilote">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="localisationPilote"
                  type="pw:entityRef"/>
    </xs:sequence>
    <xs:attribute name="email" type="xs:string"/>
    <xs:attribute name="pwd" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

Tableau 3–8 Association de type agrégation

Référence au moyen d'un attribut XML

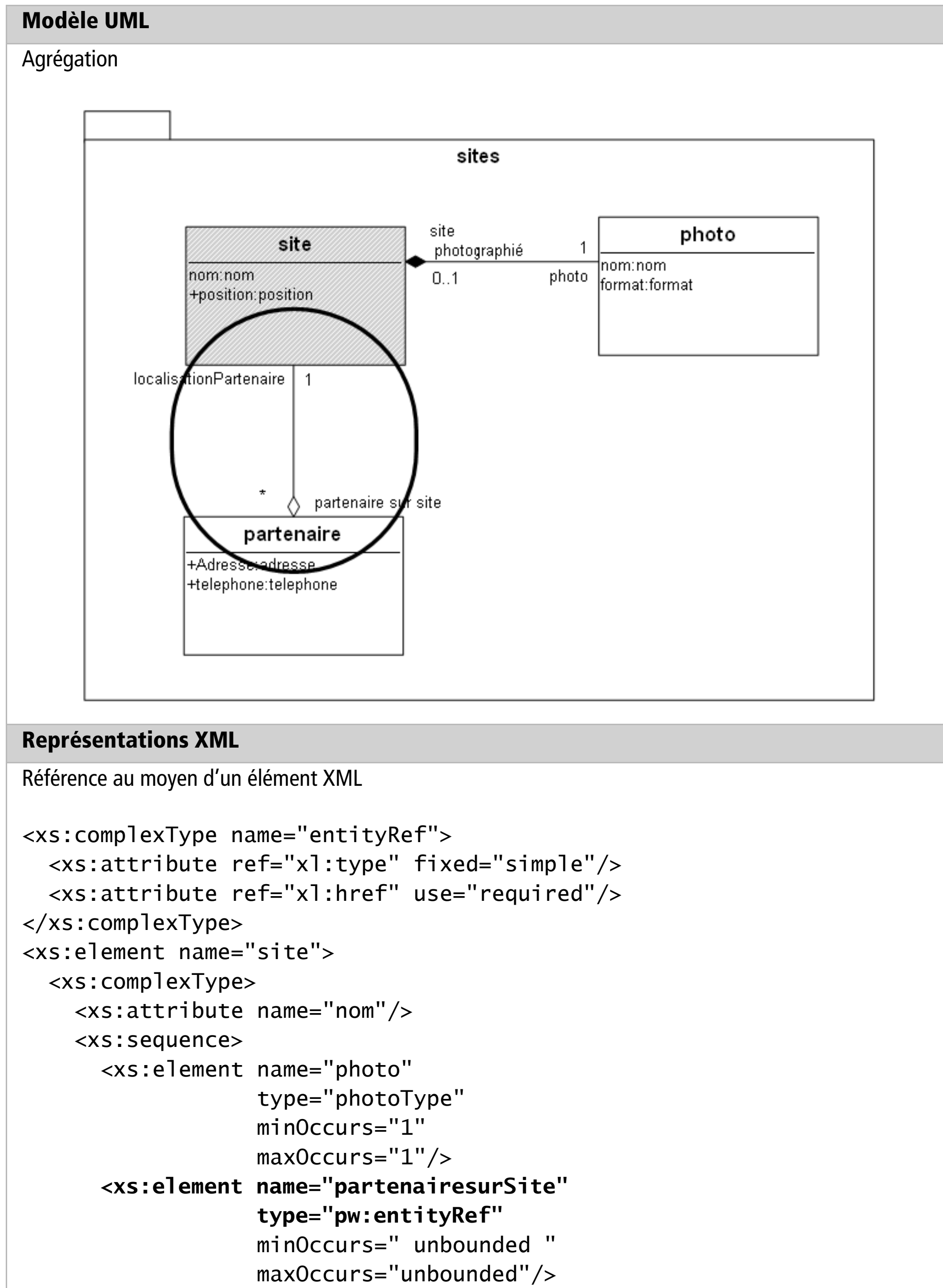
```
<xs:simpleType name="entityRef">
    <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:element name="pilote">
  <xs:complexType>
    <xs:attribute name="email" type="xs:string"/>
    <xs:attribute name="pwd" type="xs:string"/>
    <xs:attribute name="localisationPilote"
                  type="pw:entityRef"/>
  </xs:complexType>
</xs:element>
```

Prise en compte des modules de données

La prise en compte des modules de données apporte une dimension supplémentaire à la mise au point du modèle logique et permet d'affiner la mise en œuvre des première et dernière règles indiquées précédemment :

- Changement de la règle de transformation de l'association : lorsque la classe subordonnée n'est la classe entité d'aucun autre paquetage, alors l'agrégation est traitée comme une composition. Par exemple, dans PiloteWeb, on aurait pu ne pas créer le paquetage `sites` dans lequel la classe `site` est une classe entité. Dans ce cas, les types XML auxquels ont donné lieu les classes `pilote` et `aerodrome` auraient inclus hiérarchiquement la définition de `site`. En d'autres termes, les occurrences de `site` auraient été gérées sous les éléments `pilote` et `aerodrome`.
- Changement de la règle de navigabilité : lorsque le paquetage, ou module de données, d'une classe entité comprend explicitement une association où la classe entité joue le rôle de subordination, cela vient modifier la règle standard sur la navigabilité. La référence à l'agrégat doit alors être gérée. Dans l'exemple présenté dans le tableau 3-9, le paquetage `sites` de la classe `site` inclut l'association entre `site` et `partenaire`. Cela indique que la définition de site doit aussi inclure les références vers les partenaires, même si la classe `site` est la classe subordonnée dans cette association. L'objectif est de pouvoir forcer la navigation vers l'agrégat dans le contexte d'un module de données spécifique sans avoir à remettre en cause globalement cette agrégation.

Tableau 3–9 Module de données et agrégation



Modèle UML

Agrégation

Représentations XML

Référence au moyen d'un élément XML

```
<xs:complexType name="entityRef">
  <xs:attribute ref="xl:type" fixed="simple"/>
  <xs:attribute ref="xl:href" use="required"/>
</xs:complexType>
<xs:element name="site">
  <xs:complexType>
    <xs:attribute name="nom"/>
    <xs:sequence>
      <xs:element name="photo"
                  type="photoType"
                  minOccurs="1"
                  maxOccurs="1"/>
      <xs:element name="partenairesurSite"
                  type="pw:entityRef"
                  minOccurs=" unbounded "
                  maxOccurs="unbounded"/>
```

Tableau 3–9 Module de données et agrégation

```
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="pilote">
  <xs:complexType>
    <xs:sequence>
    </xs:sequence>
    <xs:attribute name="email" type="xs:string"/>
    <xs:attribute name="pwd" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

En complément de l'analyse des associations, l'analyse des modules de données apporte donc quelque lumière sur le contour d'une classe et sur la localisation des occurrences. À partir d'un même graphe de classes, il est possible de spécifier plusieurs modes de découpage. Cela est particulièrement utile lorsqu'une partie seulement du modèle de données doit être prise en compte, par exemple pour un transfert d'un paquet de données d'un système vers un autre, car la notion de module correspond, *in fine*, à la facilité avec laquelle on pourra isoler des paquets de données afin de les manipuler indépendamment du reste de l'application : c'est typiquement le cas lorsque des données doivent être échangées entre applications.

Les associations simples

Dans les associations simples, chaque classe a un rôle de poids équivalent. Cela veut dire qu'il n'y a pas de hiérarchie entre les objets et que l'association a une autonomie propre ; elle n'est pas assujettie à l'une ou l'autre des classes. En XML, on traitera ce type d'association soit comme un élément autonome contenant des références aux objets qui forment l'association, soit comme des références croisées entre les objets associés.

En optant pour la première solution d'un élément autonome pour l'association, on s'assure la plus grande flexibilité et la meilleure cohérence dans la gestion des liens. Lorsque l'association est un élément autonome, les références aux objets sont gérées conjointement sous le même élément XML. Dans l'exemple présenté au tableau 3-10, on a ajouté une nouvelle association entre la classe `pilote` et la classe `site`. Cette association indique les évaluations données par les pilotes sur la qualité des sites. L'association est porteuse de deux attributs : note et commentaire.

Le tableau 3-10 donne aussi la représentation XML de cette association sous forme d'un élément autonome en appliquant les règles suivantes :

- L'association est représentée par un élément XML. Le nom de l'élément est celui de l'association. Dans l'exemple donné par le tableau 3-10, ce nom est `evaluationsite`.
- Un élément chapeau, réunissant toutes ces associations, gère la collection des occurrences d'association. Par codification, cet élément a pour nom le nom de l'association suffixé d'un « s ». Dans notre exemple, il s'agit de l'élément `evaluationsites`.
- Chaque rôle de l'association est représenté par un sous-élément de l'élément représentant l'association. Dans notre exemple, `evaluationsite` comprend les éléments `evaluateur` et `siteévalué` correspondant respectivement au rôle de la classe `pilote` et à celui de la classe `site`.
- Le type des éléments représentant les rôles est `entityRef`, ou tout autre mécanisme permettant le référencement d'élément XML.

Tableau 3–10 Association autonome

Modèle UML

Association simple

Pilote + Préférence
pilote
+nom:nom
+email:email
+pwd:pwd
*
localisationPilote
piloteLocalisé
1
site
évaluateur
Evaluation site
site évalué
*
*
Evaluation site
+commentaire
+note

Représentations XML

Association sous forme d'un élément autonome

```
<xs:element name="evaluationsites">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="evaluationsite"
                  maxOccurs="unbounded">
        <xs:complexType>
```

Tableau 3–10 Association autonome

```
          <xs:sequence>
            <xs:element name="evaluateur"
                        type="pw:entityRef"/>
            <xs:element name="siteévalué"
                        type="pw:entityRef"/>
            <xs:element name="note"
                        type="xs:integer"/>
            <xs:element name="commentaire"
                        type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

On peut noter que la suppression d'un élément `evaluationsite` entraîne nativement celle des éléments de référence aux objets sites et pilotes, ce qui simplifie la gestion de l'intégrité référentielle. En revanche, le parcours des références est plus complexe : pour trouver les pilotes évaluateurs d'un site, il faut rechercher tous les éléments `evaluationsite` ayant une référence `siteévalué` correspondant au site recherché. Pour chaque élément `evaluationsite` trouvé, il faut ensuite rechercher les éléments `pilote` à partir de la référence donnée par l'élément `evaluateur`.

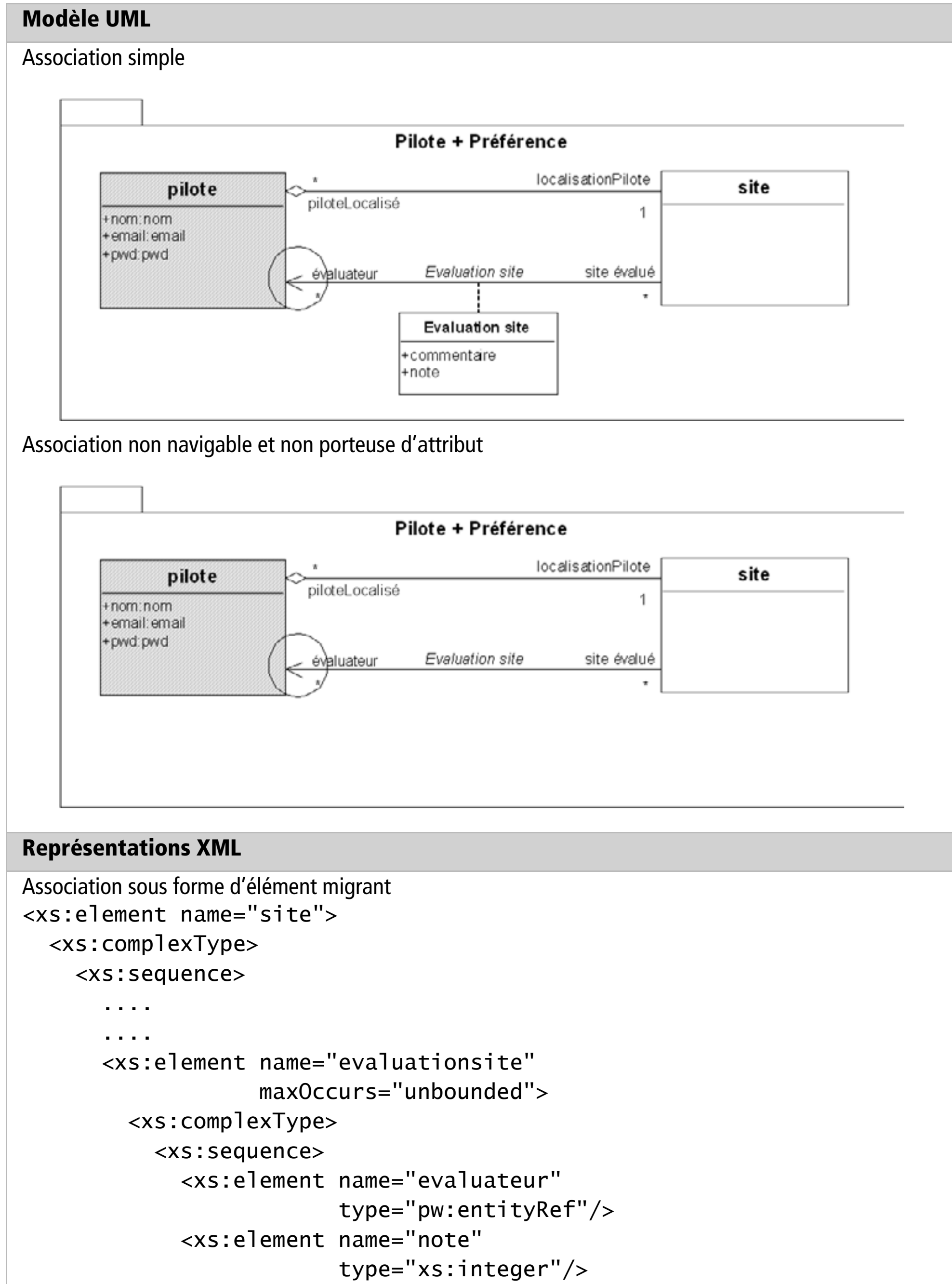
```
site -> evaluationsite.siteévalué -> evaluationsite.evaluateur -> pilote
```

Certains modèles XML avancés, comme TopicMap, considèrent les associations comme des éléments à part entière. Ces modèles sont présentés dans le chapitre 10.

Cependant, la difficulté de mise en œuvre du parcours des associations conduit le plus souvent à décider de leur orientation physique. Un seul des rôles est alors déclaré physiquement navigable, ce qui se manifeste, dans la notation UML, par une flèche du côté du rôle navigable (tableau 3-11). Les décisions sur la navigabilité peuvent être prises assez tôt dans la conception des modèles de données. Elles permettent de préciser une orientation du graphe des objets lorsque les caractéristiques d'orientation sémantiques (agrégation, composition) ne sont pas spécifiées.

Dans la représentation XML, on fait alors « migrer » les éléments de l'association dans l'élément dont le rôle n'est pas navigable. Lorsque l'association n'est pas porteuse d'attribut, seule la référence à l'objet navigable est migrée. Le tableau 3-11 donne la représentation XML correspondante.

Tableau 3–11 Association autonome



Modèle UML

Association simple

Association non navigable et non porteuse d'attribut

Représentations XML

Association sous forme d'élément migrant

```
<xs:element name="site">
  <xs:complexType>
    <xs:sequence>
      ....
      ....
      <xs:element name="evaluationsite"
                  maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="evaluateur"
                        type="pw:entityRef"/>
            <xs:element name="note"
                        type="xs:integer"/>
```

Tableau 3–11 Association autonome

```
            <xs:element name="commentaire"
                        type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="site">
  <xs:complexType>
    <xs:sequence>
      ....
      <xs:element name="evaluateur"
                  type="pw:entityRef"/>
                  maxOccurs="unbounded">
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Gestion des attributs

Nous avons vu dans le chapitre 2 que certains attributs du modèle de classe avaient un type simple, comme « chaîne de caractères », et que d'autres avaient un type composé, comme l'attribut `position` d'un site. Pour les attributs composés, la règle de transformation en XML est simple : ils donnent lieu à des types complexes comprenant des sous-éléments.

Pour les attributs de type simple, il est plus difficile de statuer. Quelques bonnes pratiques peuvent cependant être établies :

- Les attributs identifiants (par exemple `id`) ou candidats identifiants (par exemple le nom) sont représentés sous forme d'attributs XML.
- Les attributs textes de longueur significative, comme les descriptions ou commentaires, sont représentés par des éléments.
- Les attributs ayant une multiplicité supérieure à 1 sont représentés par des éléments.
- Pour les autres attributs, le choix de l'implémentation XML sous forme d'élément ou d'attributs ressortit à une politique globale de gestion de schéma. Une politique « conservatrice » optera pour l'utilisation d'éléments car elle permet une plus grande évolutivité. Une politique « d'économie » optera pour l'utilisation d'attributs XML, moins verbeuse que l'usage des éléments.

Le schéma logique de l'application PiloteWeb

En appliquant les règles énoncées depuis le début du chapitre à l'ensemble du modèle conceptuel de PiloteWeb, on obtient les modèles logiques et leurs correspondances en schémas XML présentés dans cette section.

Module de données « sites »

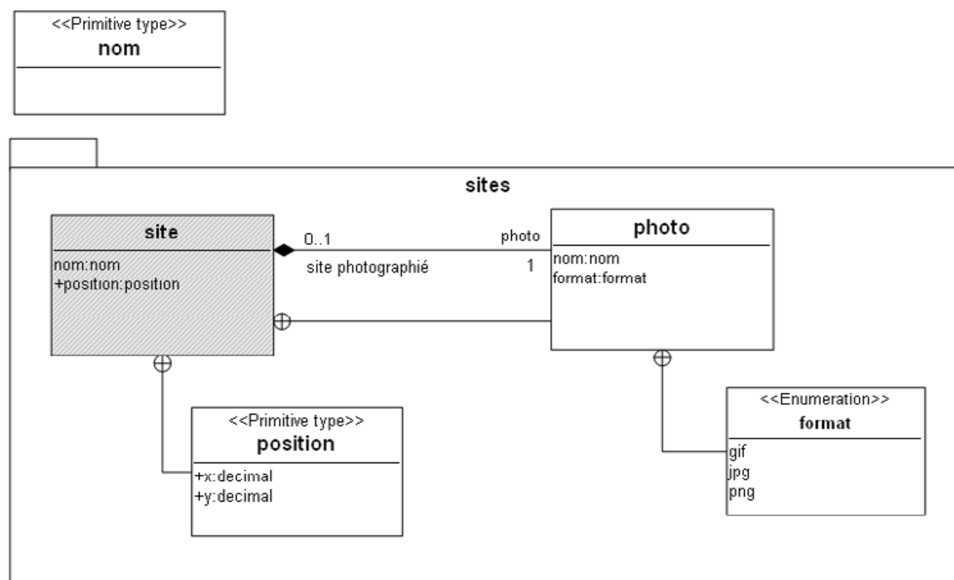


Figure 3–12 Module de données « sites »

- Il existe des occurrences indépendantes et référençables de sites. L'élément site est donc public.
- La liste des occurrences de sites est gérée sous l'élément global `sites` issu du paquetage `sites`.
- Par principe d'économie, le type de la classe `site` n'est pas publié en dehors de l'élément `site`.
- L'élément site étant référençable, un attribut `xsd:ID` lui a été automatiquement ajouté.
- L'attribut `nom` de site référence le type public `nom`. Le choix d'implémentation de PiloteWeb est d'avoir un attribut `nom` global.
- L'attribut `position` est décomposé et donne donc lieu à un type complexe.

- Le type de données position est local à la classe site. Il donne lieu à un type local de l'attribut position dans le modèle XML.
- La politique de PiloteWeb pour l'implémentation des attributs simples est l'usage des attributs XML. Les attributs x et y n'étant pas identifiants, ils donnent lieu à des attributs XML.
- La classe site a une association de composition avec la classe photo, ce qui donne lieu à un sous-élément photo dans l'élément site.
- La classe photo est déclarée comme locale à la classe site. Le type de l'élément photo est donc déclaré comme local à l'élément photo.
- L'attribut nom de photo référence le type public nom. Le choix d'implémentation de PiloteWeb est d'avoir un attribut nom global.
- La politique de PiloteWeb pour l'implémentation des attributs simples est l'usage des attributs XML. L'attribut format donne lieu à un attribut XML.
- Le type de données format est local à la classe photo. Il donne lieu à un type local de l'attribut format dans le modèle XML.

```
<xs:element name="site">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="position">
        <xs:complexType>
          <xs:sequence/>
          <xs:attribute name="x" type="xs:decimal"/>
          <xs:attribute name="y" type="xs:decimal"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="photo">
        <xs:complexType>
          <xs:attribute ref="pw:nom"/>
          <xs:attribute name="format" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="png"/>
                <xs:enumeration value="gif"/>
                <xs:enumeration value="jpg"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
```

```
      <xs:attribute name="id" type="xs:ID"/>
      <xs:attribute ref="pw:nom"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="sites">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="pw:site"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Module de données « pilotes »

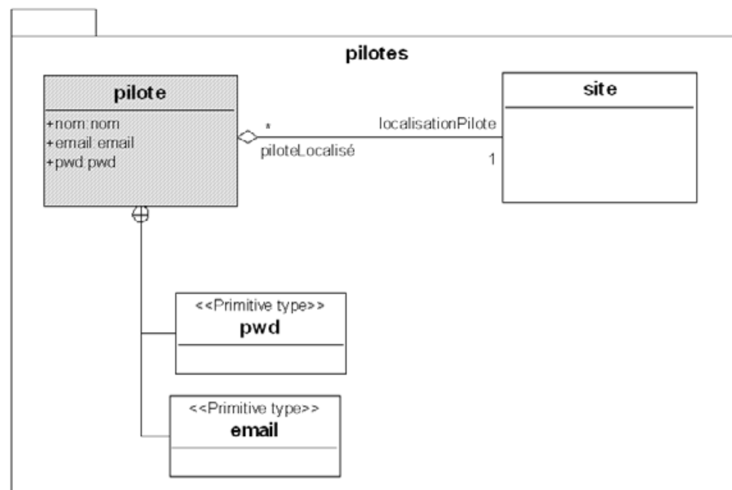


Figure 3–13
Module de données « pilotes »

- Il existe des occurrences indépendantes et référençables de `pilotes`. L'élément `pilote` est donc public.
- La liste des occurrences de `pilotes` est gérée sous l'élément global `pilotes` issu du paquetage `pilotes`.
- Par principe d'économie, le type de la classe `pilote` n'est pas publié en dehors de l'élément `pilote`.
- L'élément `pilote` étant référençable, un attribut `xsd:ID` lui a été automatiquement ajouté.
- L'attribut `nom` de `pilote` référence le type public `nom`. Le choix d'implémentation de PiloteWeb est d'avoir un attribut `nom` global.

- Le type de données `pwd` (*password*) est local à la classe `pilote`. Il donne lieu à un type local de l'attribut `pwd` dans le modèle XML.
- Le type de données `email` est local à la classe `pilote`. Il donne lieu à un type local de l'attribut `email` dans le modèle XML.
- La classe `pilote` a une association d'agrégation avec la classe `site`, ce qui donne lieu à un sous-élément `localisationpilote` dont le type est `entityRef`.

```
<xs:element name="pilote">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="localisationpilote" type="pw:entityRef"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:attribute ref="pw:nom"/>
    <xs:attribute name="pwd">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="email">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>

<xs:element name="pilotes">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="pw:pilote"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Module de données « aerodromes »

- Il existe des occurrences indépendantes et référençables d'aérodromes. L'élément `aerodrome` est donc public.
- La liste des occurrences d'aérodromes est gérée sous l'élément global `aerodromes` issu du paquetage `aerodromes`.
- Par principe d'économie, le type de la classe `aerodrome` n'est pas publié en dehors de l'élément `aerodrome`.

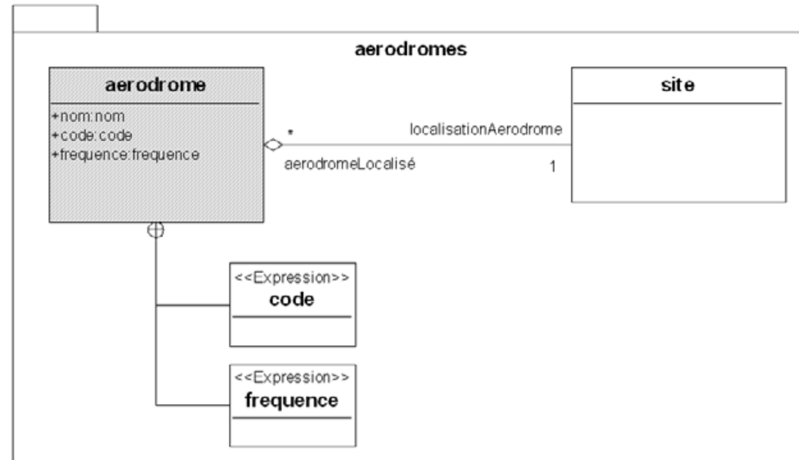


Figure 3–14
Module de données « aerodromes »

- L'élément `aerodrome` étant référençable, un attribut `xsd:ID` lui a été automatiquement ajouté.
- L'attribut `nom` d'`aerodrome` référence le type public `nom`. Le choix d'implémentation de PiloteWeb est d'avoir un attribut `nom` global.
- Le type de données `code` est local à la classe `aerodrome`. Il donne lieu à un type local de l'attribut `code` dans le modèle XML.
- Le type de données `frequence` est local à la classe `aerodrome`. Il donne lieu à un type local de l'attribut `email` dans le modèle XML.
- La classe `aerodrome` a une association d'agrégation avec la classe `site`, ce qui donne lieu à un sous-élément `localisationAerodrome` dont le type est `entityRef`.

```
<xs:element name="aerodrome">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="localisationAerodrome" type="pw:entityRef"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:attribute ref="pw:nom"/>
    <xs:attribute name="code">
      <xs:simpleType>
        <xs:restriction base="xs:string"/>
      </xs:simpleType>
    </xs:attribute>
```

```
      <xs:attribute name="frequence">
        <xs:simpleType>
          <xs:restriction base="xs:string"/>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="aerodromes">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="pw:aerodrome"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Module de données « partenaires »

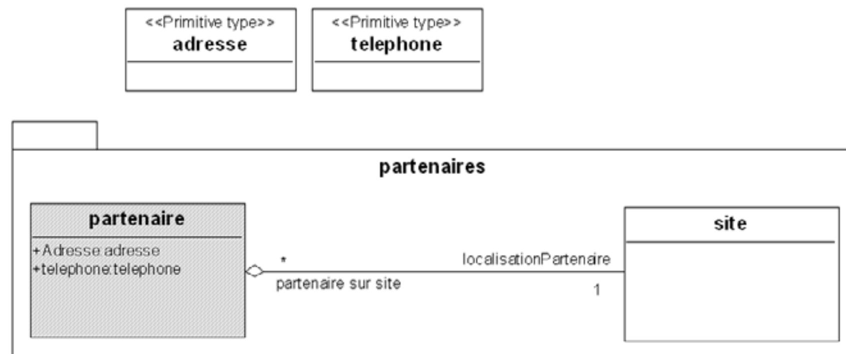


Figure 3–15
Module de données « partenaires »

- Il existe des occurrences indépendantes et référençables de `partenaires`. L'élément `partenaire` est donc public.
- La liste des occurrences de `partenaire` est gérée sous l'élément global `partenaires` issu du paquetage `partenaires`.
- Par principe d'économie, le type de la classe `partenaire` n'est pas publié en dehors de l'élément `partenaire`.
- L'élément `partenaire` étant référençable, un attribut `xsd:ID` lui a été automatiquement ajouté.
- Le type de données `adresse` est global et est utilisé comme type pour l'attribut du même nom de l'élément `partenaire` dans le modèle XML.

- Le type de données `telephone` est global et est utilisé comme type pour l'attribut du même nom de l'élément `partenaire` dans le modèle XML.
- La classe `partenaire` a une association d'agrégation avec la classe `site`, ce qui donne lieu à un sous-élément `localisationPartenaire` dont le type est `entityRef`.

```
<xs:element name="partenaire">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="localisationPartenaire" type="pw:entityRef"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:attribute name="adresse" type="pw:adresse"/>
    <xs:attribute name="telephone" type="pw:telephone"/>
  </xs:complexType>
</xs:element>
<xs:element name="partenaires">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="pw:partenaire"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Module de données « restaurants »

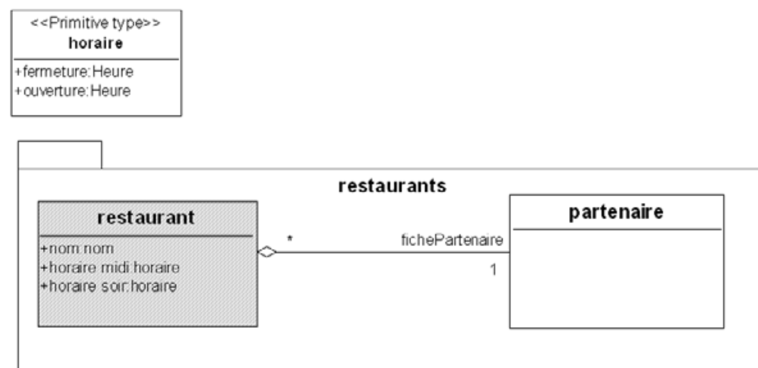


Figure 3–16 Module de données « restaurants »

- Il existe des occurrences indépendantes et référençables de restaurants. L'élément `restaurant` est donc public.

- La liste des occurrences de restaurants est gérée sous l'élément global `restaurants` issu du paquetage `restaurants`.
- Par principe d'économie, le type de la classe `restaurant` n'est pas publié en dehors de l'élément `restaurant`.
- L'élément `restaurant` étant référençable, un attribut `xsd:ID` lui a été automatiquement ajouté.
- L'attribut `nom` de restaurant référence le type public `nom`. Le choix d'implémentation de PiloteWeb est d'avoir un attribut `nom` global.
- Le type de données `horaire` étant un type composé, il donne lieu à un type complexe XML.
- Le type de données `heure` correspond au type de données standard de XML schéma : `xs:time`.
- La classe `restaurant` a une association d'agrégation avec la classe `partenaire`, ce qui donne lieu à un sous-élément `fichePartenaire` dont le type est `entityRef`.

```
<xs:complexType name="horaire">
  <xs:attribute name="ouverture" type="xs:time"/>
  <xs:attribute name="fermeture" type="xs:time"/>
</xs:complexType>

<xs:element name="restaurant">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="horaire midi" type="pw:horaire"/>
      <xs:element name="horaire soir" type="pw:horaire"/>
      <xs:element name="ficheRestaurant" type="pw:entityRef"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:attribute ref="pw:nom"/>
  </xs:complexType>
</xs:element>

<xs:element name="restaurants">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="pw:restaurant"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Module de données « loueurs »

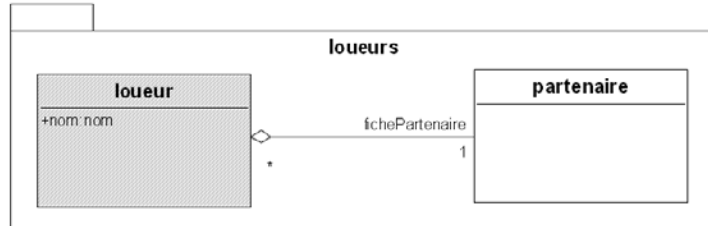


Figure 3–17
Module de données « loueurs »

- Il existe des occurrences indépendantes et référençables de loueurs. L'élément `loueur` est donc public.
- La liste des occurrences de loueurs est gérée sous l'élément global `loueurs` issu du paquetage `loueurs`.
- Par principe d'économie, le type de la classe `loueur` n'est pas publié en dehors de l'élément `loueur`.
- L'élément `loueur` étant référençable, un attribut `xsd:ID` lui a été automatiquement ajouté.
- L'attribut `nom` de loueur référence le type public `nom`. Le choix d'implémentation de PiloteWeb est d'avoir un attribut `nom` global.
- La classe `loueur` a une association d'agrégation avec la classe `partenaire`, ce qui donne lieu à un sous-élément `fichePartenaire` dont le type est `entityRef`.

```
<xs:element name="loueur">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ficheRestaurant" type="pw:entityRef"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:attribute ref="pw:nom"/>
  </xs:complexType>
</xs:element>

<xs:element name="loueurs">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="pw:loueur"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Module de données « aeroclubs »

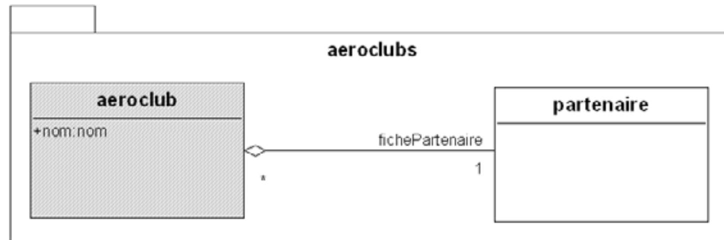


Figure 3–18
Module de données « aeroclubs »

- Il existe des occurrences indépendantes et référençables d'aéroclubs. L'élément `aeroclub` est donc public.
- La liste des occurrences d'aéroclubs est gérée sous l'élément global `aeroclubs` issu du paquetage `aeroclubs`.
- Par principe d'économie, le type de la classe `aeroclub` n'est pas publié en dehors de l'élément `aeroclub`.
- L'élément `aeroclub` étant référençable, un attribut `xsd:ID` lui a été automatiquement ajouté.
- L'attribut `nom` d'aéroclub référence le type public `nom`. Le choix d'implémentation de PiloteWeb est d'avoir un attribut `nom` global.
- La classe `aeroclub` a une association d'agrégation avec la classe `partenaire`, ce qui donne lieu à un sous-élément `fichePartenaire` dont le type est `entityRef`.

```
<xs:element name="aeroclub">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ficheRestaurant" type="pw:entityRef"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:attribute ref="pw:nom"/>
  </xs:complexType>
</xs:element>

<xs:element name="aeroclubs">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="pw:aeroclub"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Le schéma relationnel de l'application PiloteWeb

Comme nous l'avons indiqué au chapitre 2, le même modèle logique peut donner lieu soit à un schéma XML, soit à un schéma relationnel. Sans entrer dans les détails du modèle relationnel, on peut observer à figure 3-19 les différences majeures avec les correspondances effectuées pour le schéma XML.

- L'orientation du graphe des associations n'est pas prise en compte.
- L'organisation des données en modules n'est pas prise en compte.
- Les types de données complexes sont eux-mêmes transformés en liste de colonnes.

Le modèle relationnel est un modèle global à plat, à partir duquel il est difficile d'opérer par segmentation des données. Par exemple, la classe `photo` est promue au rang d'entité globale. Si une autre définition de `photo` apparaît dans le schéma, elle risque d'enter en collision avec celle donnée au titre de l'entité `site`. Dans le chapitre 4, nous aborderons plus en détail les différences entre le modèle relationnel et le modèle XML.

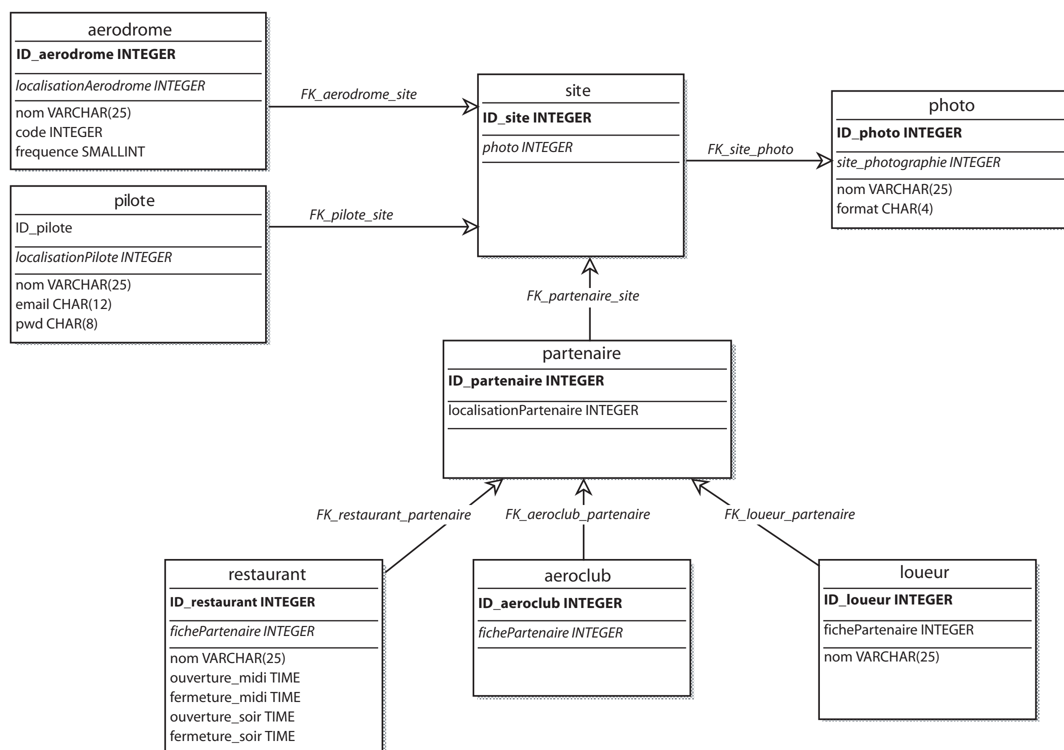


Figure 3–19 Schéma relationnel de PiloteWeb

En résumé...

Les différentes étapes d'analyse du modèle logique de données sont autant de moments indispensables au cadrage des modules qui constitueront le socle d'une architecture de données flexible et évolutive. L'analyse conjointe du regroupement des classes en paquetages et de l'orientation des associations permet de définir des règles de transformations des modèles UML vers le modèle XML. L'objectif n'est pas tant le développement de filtres bi-directionnels de conversion d'un modèle de données UML en schéma XML, et vice versa, même si, sur les cas simples, la conversion UML/XML reste possible. Identifier les classes concrètes, leur réseau de relations et les possibilités de factorisation, les dépendances qui en découlent, voici les objectifs principaux de l'analyse des modèles logiques.

4

Étape 4 - Spécifier les modèles de stockage

Un modèle physique, ce n'est rien d'autre que la manière dont les données sont physiquement stockées et identifiées.

Les modèles physiques diffèrent en effet des modèles logiques car, à ce stade de la conception, il reste à définir précisément :

- les lieux réels de stockage des documents XML et de leurs schémas associés ;
- les relations entre éléments qui dépendent du stockage physique ;
- les identificateurs d'éléments qui peuvent également dépendre du stockage physique ;
- les noms des documents XML qui peuvent être dépendants du lieu de stockage physique.

Ainsi, la représentation physique des données n'est pas nécessairement une projection directe du modèle logique. Par exemple, il peut être utile, pour des raisons fonctionnelles de temps de réponse ou autre, de découper le modèle logique en plusieurs sous-schémas. Il reste alors à créer ces nouveaux modèles et décider du lieu de stockage des documents XML qui leur correspondent.

Avec les modèles physiques, nous allons donc être amenés à :

- spécifier l'éventuel découpage du schéma logique en sous-schémas, et introduire le cas échéant de nouveaux éléments XML ;
- spécifier les URI correspondant aux espaces de noms des sous-schémas ;
- spécifier les lieux de stockage des sous-schémas et documents XML correspondants ;
- adapter les liens en conséquence.

Ainsi, les modèles physiques représentent la dispersion physique des données, traduisent des choix d'implémentation et doivent permettre de garantir une exploitation optimale des données par les programmes.

À cet effet, les représentations UML suivantes pourront être utiles : diagrammes de classes, diagrammes de composants et diagrammes de déploiement.

Réaliser un modèle physique, démarche générale

Pour définir le modèle physique de stockage, nous adoptons une démarche de conception en trois temps, que nous présentons dans cette section.

Il faut commencer par identifier les questions générales :

- Gestion d'un seul grand document XML ou de plusieurs petits ?
- Gestion des révisions à l'intérieur des documents XML ou par recopie des documents XML ?
- Stockage dans des fichiers, une base de données relationnelle ou une base de données XML ?

Ces grandes lignes étant connues, il faut ensuite construire la stratégie d'adressage des données :

- Quels noms donner aux documents XML ?
- Comment identifier les objets à l'intérieur des documents XML ?
- Quel langage de liaison mettre en œuvre (XLink, XInclude, autres) ?
- Les liens reposeront-ils sur des URL ? Des URI ? Des URN ? Des chemins d'accès physiques, logiques ?
- Comment gérer les liens entre documents en base et documents hors base ?

On terminera la conception en précisant quels éléments serviront concrètement de base aux liens en fonction des langages et modèles de liaison retenus : on choisira parmi les expressions XPointer, XPath, et requêtes XQuery, selon le modèle physique établi.

Définir la manière dont les données seront physiquement stockées

Choisir une forme de stockage

À ce stade, nous n'avons abordé le problème de la modélisation que sous le seul angle fonctionnel du système. Les relations entre les données avaient comme objectif d'exprimer la logique de l'application. Or, dans notre hypothèse d'un système « tout XML », ces mêmes données doivent être stockées dans des documents XML. C'est pourquoi nous allons aborder ici la question du modèle de stockage à utiliser.

Selon la façon de considérer le schéma XML représentant la vue logique, XML offre quatre possibilités :

1. Il est conservé en l'état et les données sont entièrement contenues dans un seul et même document XML.
2. Il est conservé en l'état mais les données donnent naissance à plusieurs documents XML du même type.
3. Il est subdivisé et les données sont réparties dans autant de documents XML qu'il y a de sous-schémas.
4. Il est subdivisé et les données sont réparties dans plusieurs documents XML de même type pour chacun des sous-schémas.

Le tableau 4-1 résume ces quatre situations.

		Nombre de schémas	
		1	**Plusieurs**
Nombre de documents XML par schéma	**1**	La totalité des données de l'application est stockée dans un seul document XML.	Il y a plusieurs schémas et un document XML de stockage des données par schéma.
	Plusieurs	Il n'y a qu'un seul schéma, mais plusieurs documents XML de stockage contiennent des données conformes à ce schéma.	Il y a plusieurs schémas et, pour chacun, plusieurs documents XML de stockage des données.

À chaque cas correspondent des applications types que nous allons brièvement décrire.

Un seul schéma et un seul document

Dans ce cas, les données de l'application sont stockées dans un seul gros fichier XML. Cela n'est évidemment possible que si le volume de données reste acceptable

au regard des temps de réponse et si un seul schéma XML suffit à satisfaire les besoins de l'application. Enfin, il faut savoir que, dans un tel cas, les mises à jour de la base ne pourront se faire que l'une après l'autre.

Des exemples :

- les données échangées entre deux systèmes informatiques ;
- base contenant la totalité d'un manuel de maintenance d'avion.

Quelques caractéristiques :

- Le modèle de stockage physique est égal au modèle logique.
- Le temps de montée du document en mémoire (DOM) peut être long, comme toutes les opérations de manipulation du document. Le système de stockage doit pouvoir intervenir directement sur le DOM persistant.
- Le système de stockage peut être un simple fichier.
- Il n'y a pas de risque de conflit de noms de fichiers puisque, par définition, il n'y en a qu'un seul. Le nom du document XML qui contient les données peut être déterminé et repéré bien à l'avance.

Un seul schéma et plusieurs documents

Dans ce cas, la totalité des données tient dans un seul type de document XML ; dans ce dernier toutefois, plusieurs documents sont nécessaires pour contenir toutes les données.

Des exemples :

- Le cas typique est celui des documents XML figurant des documents papier. Le modèle XML représente alors l'ensemble des documents (papier) d'un même type, d'une même famille. On aura par exemple un modèle XML pour l'ensemble des modèles de lettres, un autre pour l'ensemble des modèles de fax, etc. Il y aura autant de documents XML que de documents papier produits.
- Un autre cas typique est celui de données qui représentent par exemple des bons de commande : un seul schéma XML correspond à n bons de commande.

Quelques caractéristiques :

- Les noms des documents XML produits devront forcément être différents ; une forme lexicale générique de production de noms de fichiers ne peut être décidée une fois pour toutes.
- Ces ensembles de documents sont sensibles aux liens. Les noms des fichiers ne sont pas aisés à modifier si des liens inter-documents existent.

Plusieurs schémas et un seul document par schéma

Ce cas est caractéristique des applications orientées données, comme PiloteWeb. Les documents XML trouvent alors très bien leur place au sein d'une base de données XML. Les données sont réparties dans différents documents XML ayant chacun son propre schéma. Chaque document XML contient la totalité des données d'un même schéma.

Par exemple, dans PiloteWeb, le document XML `pilotes.xml` contient la totalité des données spécifiques aux pilotes, définies par le schéma `pilote.xsd`. On aurait pu décider que chaque pilote disposerait de son propre fichier de données mais la conception a décidé qu'elles seraient toutes réunies dans un seul et même fichier.

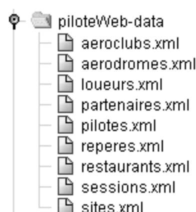


Figure 4–1
Toutes les données de PiloteWeb tiennent dans 9 documents XML de schémas différents.

On en trouve des exemples typiques dans les applications de gestion de pièces de rechange, de commerce électronique, etc.

Quelques caractéristiques :

- Les noms des documents XML de stockage sont connus à l'avance.
- Ils peuvent être contenus dans un même répertoire.
- Un tel modèle fait penser aux tables de bases de données classiques : le rôle de chaque document XML peut être assimilé à celui d'une table. Toutefois, la structure interne d'un document XML est à la fois infiniment plus riche et souple que celle d'une table.

Plusieurs schémas et plusieurs documents par schéma

Dans ce cas, il faut plusieurs documents XML pour stocker les données d'un même type. Cela se produit donc quand on est obligé de dupliquer les documents XML d'un même type :

- soit parce qu'il y en a trop et que les documents XML seraient trop volumineux ;
- soit parce qu'il serait inconcevable de garder toutes les données dans un seul document XML : c'est le cas des documents « papier » dématérialisés.

Un exemple caractéristique est celui des documents « papier » découpés en chapitres et/ou sections. Les chapitres et les sections sont tous d'un même type, mais il est inconcevable de ne pas avoir un document XML par chapitre ou section.

Afin que plusieurs documents XML d'un même type puissent cohabiter dans un même répertoire, le nom des fichiers les contenant doit obligatoirement varier d'un document XML à un autre.

Quelques exemples :

- Toute documentation qui est faite sur le principe de l'assemblage de modules structurés.
- Tout document XML contenant des données que l'on a préféré répartir dans plusieurs fichiers. Par exemple, on peut décider, pour gérer les villes françaises, d'avoir 26 documents XML d'un même type : un par lettre de l'alphabet.

Quelques caractéristiques :

- Les noms des fichiers contenant les documents XML peuvent ou non être connus à l'avance ; quand ce n'est pas le cas, on définira une forme lexicale générique permettant de produire ces noms.
- Tout lien doit obligatoirement tenir compte du nom précédent. En fonction de la stratégie de gestion des révisions choisie, les liens devront également être révisés en cas de révision d'un document XML par copie.
- La recherche d'une information devra balayer plusieurs documents XML. On pourra procéder à l'indexation des documents XML.

Choisir une solution de stockage

Comme nous l'avons rappelé dès l'introduction générale de cet ouvrage dans la section « XML au cœur des systèmes d'information », l'une des spécificités de XML est la diversité des possibilités de stockage : direct et en clair sous la forme de fichiers du système d'exploitation, éclaté dans les tables d'une base de données relationnelle, ou encore sous la forme binaire et indexée d'une base de données XML.

Dans les sections suivantes, nous présentons les grandes caractéristiques de ces différentes solutions de stockage.

Stockage dans le système de fichiers

La représentation la plus commune des documents XML est le fichier. Le document XML est connu pour être un fichier échangeable facilement. En effet, rien n'est plus facile que de stocker un fichier XML.

La difficulté ne tient pas véritablement au stockage mais à ses conséquences :

- Quid des possibilités de recherche ?

- Quid des liens ?
- Quid des possibilités de recoupement de données entre plusieurs documents XML ?

En ce qui concerne le premier point, il est clair que les systèmes d'exploitation actuels ne sont pas adaptés au stockage XML et ne fournissent aucune fonction particulière d'indexation et de recherche de l'information, et encore moins des services de requêtes de type XQuery.

En revanche, toutes les recommandations et normes dérivées de XML font exclusivement appel aux URI et URL comme mécanismes standards d'adressage. Une conséquence immédiate est que les documents XML peuvent être stockés à n'importe quel point d'un réseau accessible par une URL, à la manière des pages HTML.

Le stockage dans le système de fichiers est donc très simple, mais aussi très pauvre. La recherche d'information est lente et il n'existe pas d'autre solution que de charger les documents XML en mémoire : sans possibilité de requêter sur les documents XML ainsi stockés, l'affichage même d'un mot nécessite de charger la totalité du document XML en mémoire ou d'utiliser des programmes de filtrage. Les systèmes de fichiers sont également très faibles en gestion des autorisations d'accès, contrôle des révisions et sécurité des transactions.

Stockage dans une base de données relationnelle

Le stockage des documents XML dans les bases de données relationnelles n'est possible que dans un seul cas de figure : lorsqu'il n'y a qu'un seul schéma et que les tables contiennent toutes les données d'un même type. Cela revient à dire que n tables représentent un schéma et que les documents valides par rapport à ce schéma sont éclatés dans ces n tables. Pour qu'un modèle relationnel représente plusieurs schémas, il faudra obligatoirement donner un préfixe aux éléments de chaque schéma (sinon, les éléments de même nom seront confondus).

Le schéma XML de stockage doit être cohérent avec ce type de stockage. Par exemple, il ne doit contenir aucun modèle mixte. Cela fait une contrainte supplémentaire forte à ajouter à celles déjà relevées au tout début de ce chapitre.

Avec le modèle relationnel, la dispersion des données dans différentes bases n'est plus possible : quand les éléments sont répartis dans des tables, le concept même d'URL disparaît ; les liens écrits avec Xpointer et XPath n'ont plus de sens direct. Sinon impossible, leur trouver une correspondance est dans le meilleur des cas coûteux en temps de traitement. Dans le cas d'école exposé dans ce chapitre, nous verrons comment Oracle XML DB résout cette question : les documents XML sont simultanément stockés sous forme d'une hiérarchie de fichiers et éclatés dans des tables conformément aux principes théoriques que nous avons déclinés.

Pour mettre en correspondance les modèles XML et relationnel, deux manières d'aborder le problème peuvent être mises en œuvre. L'une vise à rendre les mondes XML et relationnels interchangeables tandis que l'autre tend à les rendre uniformes :

- On peut stocker un document XML indépendamment de son schéma. Le modèle relationnel est alors la simple reproduction de la forme des documents XML : des éléments emboîtés les uns dans les autres sans typage particulier. On peut stocker dans une seule base physique des documents XML ayant des schémas différents.
- Ou bien on stocke le document XML selon un modèle relationnel adapté à sa DTD ou son schéma XML. La cohabitation de documents XML issus de différentes DTD ou schémas XML n'est plus possible, de même que l'utilisation d'éléments de même nom qui sont de types différents. La seule manière de contourner cette difficulté est d'utiliser l'artifice qui consiste à préfixer les noms des tables et des colonnes par le nom de leur modèle.

L'intérêt du stockage des données XML dans une base de données relationnelle est pourtant réel pour les entreprises :

- réutilisation d'un système déjà existant et éprouvé ;
- base installée importante ;
- disponibilité de la compétence ;
- qualité éprouvée dans les domaines de la gestion des transactions et la sécurité.

Cependant, le modèle XML organise les informations sous forme d'arbre tandis que le modèle relationnel les organise dans des tables. Il n'y a donc pas de correspondance directe entre ces deux modèles et, par conséquent, passer d'un modèle à l'autre requiert une certaine vigilance : tout n'est pas possible. L'information du modèle relationnel doit être organisée de manière à refléter la nature hiérarchique de XML.

D'autres problèmes persistent quand on veut stocker des documents XML dans du relationnel :

- Qu'arrive-t-il quand on veut gérer les données en révision ? Dans ce cas, les relations père-fils deviennent dépendantes du numéro de révision et un élément (donc potentiellement une table) peut appartenir à plusieurs révisions et avoir plusieurs numéros d'ordre…
- Aucun des modèles relationnels possibles ne permet d'offrir autant de degré de liberté de navigation dans les documents stockés que le modèle XML. Les cibles XLink et XPointer n'ont aucune correspondance.
- Le stockage des modèles mixtes n'est pas possible.
- L'usage des espaces de noms pourrait compliquer davantage le stockage dans les tables.

Nous pensons que la mise en correspondance de données XML avec des modèles relationnels est fondamentalement réductrice et n'a d'intérêt que dans un seul cas : lorsqu'il faut fondre les données XML reçues dans des données relationnelles, et ce sachant que l'on ne devra jamais ressortir ces données sous une forme identique au fichier XML utilisé pour l'import, ni avoir un système ouvert à n'importe quel schéma XML ou DTD.

Il est exclu de stocker des documents XML représentant des documents (qu'il s'agisse de documents papier ou de pages HTML) dans des systèmes relationnels. Il est également illusoire de tenter de stocker dans des tables les documents XML car il faudrait gérer des révisions au niveau des nœuds ; il est enfin vain de penser conserver les possibilités de navigation dans les données et de lancement de requêtes qu'offre XML : la puissance de XQuery ne se retrouve pas dans SQL ni dans une quelconque adaptation de XQuery au relationnel.

Dans cette section, nous allons expliquer les trois méthodes de transposition XML-Relationnel disponibles :

- projection d'un document bien formé ;
- projection d'une DTD ;
- projection d'un schéma XML.

Le premier cas consiste à élaborer un modèle relationnel générique capable de recevoir n'importe quel document XML. Dans le deuxième cas, il s'agit de construire le modèle relationnel à partir de la lecture fidèle d'une DTD, et, dans le dernier cas, on élabore le modèle à partir d'un schéma XML.

Il faut avant tout se détacher de l'aspect textuel d'un document. Après tout, un document XML n'est qu'une manière parmi n de sérialiser l'information qu'il contient. Pour aborder le problème de la transposition en relationnel, nous allons devoir nous intéresser à ce que représente réellement le modèle XML, à savoir des objets relatifs :

- aux données : éléments, espaces de noms, attributs et caractères ;
- aux documents eux-mêmes : le prologue, les instructions de traitement, les commentaires, les définitions de notations et d'entités.

Dans le cadre de cette section, seuls les objets de la première catégorie nous intéressent. Dans la réalité, il faudrait vérifier que la base de données envisagée supporte effectivement les autres objets.

Construction de structures arborescentes dans le modèle relationnel

Pour stocker un arbre dans un système de gestion de données relationnelles, on réalise une mise en correspondance, ou *mapping*, entre les informations de l'arbre et le schéma de la base de données. Une pratique répandue consiste à créer une table par nœud, ou élément, de l'arbre. Les relations de parenté entre les éléments sont alors

représentées par des clés : une clé pour l'élément parent, une référence à cette clé (clé étrangère) pour chaque élément enfant.

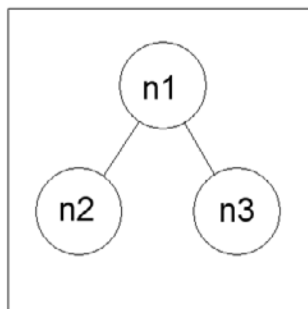


Figure 4–2
Représentation la plus générique d'un document XML sous forme relationnelle

Dans l'exemple de la figure 4-2, l'élément `n1` a deux enfants `n2` et `n3`. Nous allons utiliser deux tables, `Table 1` pour l'élément `n1` et `Table 2` pour les éléments `n2` et `n3`. Il est important de remarquer que ces tables contiennent deux types d'informations :

- celles propres à l'élément ;
- celles propres à la structure.

Celles propres à l'élément ont été placées dans une colonne intitulée `Informations auxiliaires`. Il s'agit de ses attributs, de son contenu textuel et potentiellement d'informations complémentaires comme son type.

Celles propres à la structure ont été réparties de la manière suivante :

- dans les colonnes `ID` de `Table 1` et `Table 2` qui contiennent une clé par élément ;
- dans la colonne `PARENT_ID` de `Table 2` qui référence le père de chaque élément de la table `Table 2` ;
- dans la colonne `INDEX` qui permet d'ordonner les enfants des nœuds de la table `Table 1`.

La figure 4-3 fournit la représentation en UML de ce modèle tandis que la figure 4-4 représente les tables correspondantes.

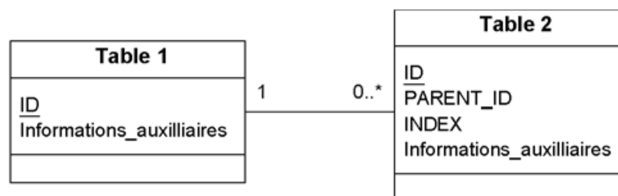


Figure 4–3
Diagramme logique du modèle générique

Figure 4–4
Tables relationnelles de notre modèle générique

TABLE-1	
ID	**Informations auxiliaires**
n1	

TABLE-2			
ID	**ID_PERE**	**INDEX**	**Informations auxiliaires**
n2	n1	0	
n3	n1	1	

Une représentation plus conceptuelle de ce modèle montrerait explicitement la relation parent-enfant. Pour cela, l'attribut migrant `PARENT-ID` deviendrait un rôle dans une association et l'attribut `index`, qui sert à gérer l'ordre d'apparition des éléments, deviendrait la caractéristique `ordonné` telle qu'elle apparaît à la figure 4-5.

> **VOCABULAIRE Attribut migrant**
>
> Un attribut est qualifié de migrant pour indiquer qu'il représente une information atomique déjà présente dans une autre table. En particulier, il est intéressant de mettre en avant de tels attributs quand une relation particulière unit les deux informations.

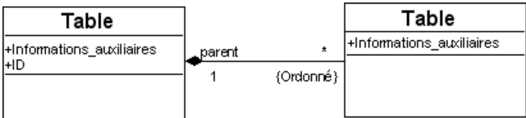


Figure 4–5
Diagramme conceptuel du modèle générique

Il n'existe pas une façon unique d'assurer le stockage d'un document XML dans un système de gestion de données relationnelles. Par exemple, on peut choisir de regrouper les informations d'un élément et ses attributs dans une seule table ou, au contraire, de les répartir sur plusieurs tables. Parmi toutes les possibilités, on distingue deux cas extrêmes :

- La totalité du document XML est ramené à une seule unité d'information ; typiquement, le document textuel est stocké dans une colonne d'une table.
- Chaque élément du document XML est stocké dans une table indépendante et des index permettent de reconstruire le document.

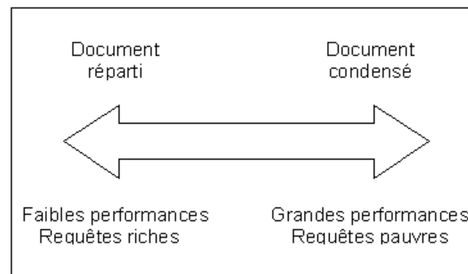
Le choix de l'un ou l'autre cas de figure a des conséquences importantes :

- Sur les performances : dans le premier cas, retrouver le document est extrêmement facile puisqu'une seule requête est nécessaire. Dans le second cas en revanche, l'information du document étant éclatée sur un grand nombre de tables, le nombre de requêtes nécessaires à la reconstitution du document va être très important.
- Sur la recherche de documents ou de fragments de documents selon des critères spécifiques : le premier cas est très pauvre et ne permet de faire des recherches que par mots-clés ou par reconnaissance de motifs textuels ; au contraire, le second

cas permet de faire des recherches bien plus riches puisque chaque élément peut être trouvé directement. La base de données peut alors créer des index des noms et valeurs des éléments, attributs et identificateurs uniques.

La figure 4-6 a pour objectif de montrer que l'alternative entre un stockage condensé et un stockage réparti est un choix antinomique.

Figure 4–6
Le paradoxe du stockage relationnel de documents XML



Dans les sections suivantes, nous allons présenter plus en détail les possibilités de modélisation qui s'offrent pour définir des schémas relationnels adaptés au stockage de documents XML. Cette présentation n'est pas exhaustive, car il y a, comme nous l'avons dit, plusieurs façons de procéder. Les choix que nous avons faits se fondent sur plusieurs critères :

- La manière dont l'information est repartie dans le schéma relationnel.
- La technique utilisée pour stocker les informations relatives à la structure arborescente.
- Les données ajoutées au schéma relationnel pour conserver l'information contenue dans le document XML.

Chaque approche sera analysée de la même manière : après avoir exposé le modèle, nous en présenterons les forces et faiblesses.

L'approche Document Object Model (DOM)

Cette approche utilise la représentation d'un document sous sa forme DOM : le document XML va être reproduit dans la base de données par une extraction des informations réalisée grâce aux API de DOM, qui définissent comment accéder aux informations d'un contenu dans un document XML. Dans ce cas, c'est la structure définie par DOM qui construit le schéma relationnel.

Si on ne considère que les unités d'information *Document*, *Element*, *Attribut* et *Text*, la représentation UML du schéma relationnel est représentée par la figure 4-7.

Lors de l'insertion d'un document XML dans la base de données, il faut tout d'abord en obtenir la vue DOM qui servira à insérer les informations relatives aux nœuds

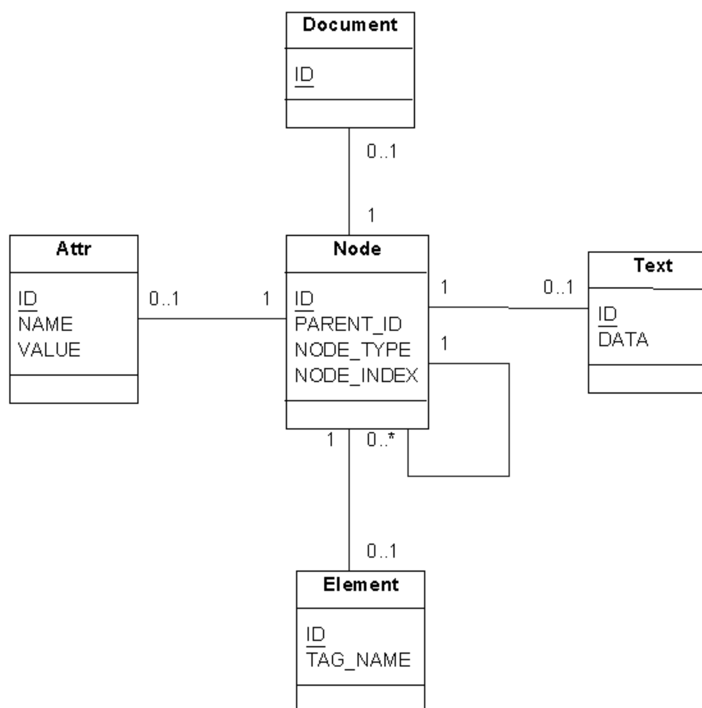


Figure 4–7
Représentation UML du schéma relationnel dans le cas de l'approche DOM

dans les bonnes tables. Chaque élément est décrit par deux tables et tous ont en commun la table `Node` (voir figure 4-7) qui contient les informations relatives aux relations parents-enfants de la structure ; c'est donc elle qui porte la structure d'arbre du document.

- L'attribut `PARENT_ID` est une clé étrangère de cette table qui pointe sur elle-même.
- L'attribut `NODE_TYPE` identifie le type du nœud et désigne donc implicitement la table où se trouve le reste de l'information appartenant à ce nœud.
- L'attribut `NODE_INDEX` est la position du nœud par rapport à sa fratrie. Cette table permet de conserver l'ordre initial des éléments du document.

Cette approche présente les points forts suivants :

- Son aspect générique. Le modèle n'est lié à aucun document XML en particulier et permet de mettre dans la base n'importe quel document XML.
- Simplicité de mise en œuvre, après construction du DOM, chaque élément est stocké au moyen d'un simple parcours de l'arbre.
- Le modèle de contenu mixte est pris en compte par le système puisque dans DOM tout contenu textuel fait l'objet d'un nœud identifiable au même titre qu'un sous-élément.

En voici toutefois les points faibles :

- L'utilisation de l'interface SAX n'est pas possible ; il faut construire le modèle DOM du document en mémoire avant de l'insérer. C'est donc pénalisant en termes de vitesse de traitement et d'occupation mémoire.
- Le document étant entièrement réparti dans le schéma relationnel, le nombre de requêtes nécessaires pour manipuler un document est proportionnel au nombre de nœuds concernés. En particulier, l'insertion d'un élément nécessite deux requêtes pour l'élément lui-même et deux requêtes pour chacun de ses attributs, ce qui est très pénalisant en termes de performances.

L'approche DTD

Cette approche se base sur le fait que le modèle de contenu défini par une DTD permet de connaître à l'avance les attributs portés par les éléments. Par conséquent, il est possible d'associer à chaque élément une table stockant non seulement l'élément mais encore ses attributs. En général, le nom de la table est le nom de l'élément, mais pas obligatoirement. Les attributs forment les colonnes de la table. Ainsi, chaque élément défini dans la DTD est associé à une table du modèle relationnel. Outre les colonnes réservées aux attributs, on va trouver dans chaque table deux champs : l'un va servir de clé et l'autre de clé étrangère indiquant l'élément père (à l'exception de la table associée à l'élément racine de la DTD). C'est le modèle logique qui structure le schéma relationnel.

Nous allons fournir un exemple de cette approche à partir d'un cas concret.

Supposons la DTD suivante :

```
<!ELEMENT bibliotheque (ouvrage*)>
<!ATTLIST bibliotheque nom #PCDATA #IMPLIED>
<!ELEMENT ouvrage (auteur+)>
<!ATTLIST ouvrage titre #PCDATA #IMPLIED>
<!ELEMENT auteur (#PCDATA)>
```

et son instance valide :

```
<bibliotheque nom="">
   <ouvrage titre="" annee="">
      <auteur>...</auteur>
      ...
      <auteur>...</auteur>
   </ouvrage>
   ...
```

```
   <ouvrage titre="">
      <auteur>...</auteur>
      ...
      <auteur>...</auteur>
   </ouvrage>
</bibliotheque>
```

Le schéma relationnel est construit à partir de la DTD. La première version de ce schéma est construite en utilisant des clés naturelles qui sont tout simplement les valeurs des noms des éléments, comme le montre le diagramme UML de la figure 4-8.

Dans l'approche DTD, il n'y a pas, à proprement parler, de mécanisme d'insertion des documents. La seule fonction assurée par l'interface est la lecture des données contenues dans le document XML et leur recopie dans les bonnes tables. Une utilisation typique des données ainsi stockées est leur exploitation pour affichage via des requêtes SQL sans aucune tentative de reconstruction d'un quelconque document XML (et encore moins un document XML égal à celui ayant servi à renseigner la base).

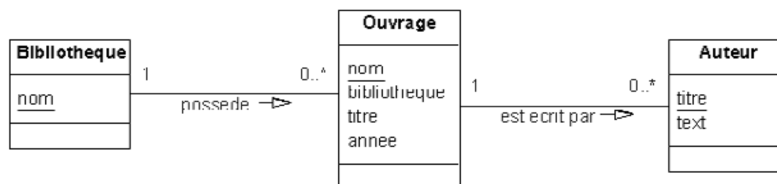


Figure 4–8
Diagramme UML du schéma relationnel utilisant des clés naturelles dans l'approche DTD

La seconde version utilise, quant à elle, des clés artificielles qui seront générées par le système de persistance lors de l'insertion du document.

> DÉFINITION **Clé artificielle**
>
> Contrairement aux clés naturelles, les clés artificielles ne possèdent pas de sémantique particulière. Elles sont souvent utilisées dans les cas suivants :
> - Quand aucun champ de la relation n'apparaît comme bon candidat pour être une clé, ce qui est le cas dans notre exemple.
> - Pour éviter d'avoir des clés étrangères sur plusieurs colonnes, souvent compliquées à gérer.
> - Quand l'identifiant d'une relation est susceptible de changer, il est alors préférable d'utiliser une clé artificielle. Ainsi, les contraintes d'intégrité ne changeront pas lors de changements d'attributs de la relation.
>
> Généralement, les clés artificielles sont produites par un générateur qui en garantit l'unicité. Les systèmes de gestion de bases de données modernes fournissent généralement une fonction de génération de telles clés.

Ce second schéma est représenté à la figure 4-9.

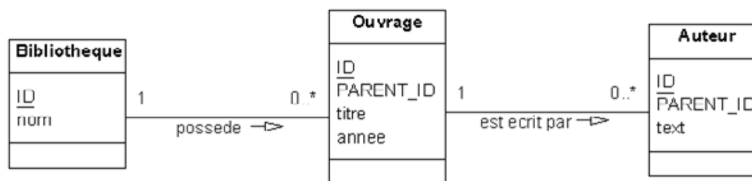


Figure 4–9 Diagramme UML du schéma relationnel utilisant des clés artificielles dans l'approche DTD

Les points forts de cette approche sont les suivants :

- La modélisation est intuitive puisqu'elle suit le modèle logique défini au départ dans la DTD. Le modèle logique est une évidence et apparaît à la première lecture du schéma relationnel.
- Il est aisé d'écrire des requêtes : c'est pratiquement les mêmes requêtes que l'on écrirait directement sur le modèle relationnel.
- L'insertion de données nécessite un nombre de requêtes moindre que dans l'approche DOM. Dans ce cas, l'insertion d'un élément ne requerra qu'une requête et il en résultera donc de meilleures performances.
- Cette approche est très peu intrusive ; les données nécessaires pour conserver la structure d'arbre sont faibles ou inexistantes.

Néanmoins, voici ses points faibles :

- Chaque modèle relationnel est spécialisé pour un type de document particulier et doit donc être écrit spécialement pour une DTD. Des outils toutefois permettent d'automatiser ce processus.
- L'ordre des éléments est perdu. Il reste possible de définir un attribut supplémentaire de table pour ordonner les résultats obtenus lors des requêtes selon l'ordre initial du document. Avec l'approche DTD cependant, si l'on souhaite transformer des données XML entrantes en pures données SQL, l'ordre initial n'importe pas quand les données sont exploitées par des requêtes globales qui portent sur l'ensemble des données.
- Ce modèle ne marche pas pour les éléments qui ont un contenu mixte.

L'approche XML Schema

L'utilisation des schémas XML pour produire une correspondance entre un document XML et un système de gestion de données relationnelles diffère assez nettement de celle d'une DTD. Tout d'abord, une DTD est basée sur la définition d'éléments, chaque élément établissant son modèle de contenu et ses attributs, alors qu'un schéma introduit la notion de type et est ainsi susceptible de casser complètement la relation biunivoque qui existait dans les DTD entre un nom d'élément et un modèle de contenu.

Dans XML Schema, les types peuvent être simples ou complexes.

Les types simples sont utilisés pour contraindre un élément ou un attribut à avoir un contenu textuel d'un type donné. Par exemple :

```
<xs:element name="exemple" type="xs:integer">
```

signifie que le contenu textuel de l'élément `exemple` sera forcément un nombre et devra être conforme à la forme lexicale définie pour le type `integer` dans le tome II de la recommandation XML Schema ; par exemple :

```
<exemple>157<exemple>
```

Les types complexes ne s'utilisent que sur des éléments et permettent de leur adjoindre des attributs et d'avoir un modèle de contenu comportant des éléments, par exemple :

```
<xs:complexType name="mon_type">
 <xs:element name="element_1" type="xs:string"/>
 <xs:element name="element_2" type="xs:integer"/>
 <xs:attribute name="att_1" type="xs:integer"/>
</xs:complexType>
<element name="exemple" type="mon_type"/>
```

Ce modèle signifie que le contenu de l'élément `exemple` est composé d'un attribut `att_1`, lui-même de type `integer`, et de deux sous-éléments, `element_1` et `element_2`, respectivement de type `string` et `integer`.

Les schémas XML Schema empruntent plusieurs notions à la programmation orientée objet, comme l'héritage et la notion de type abstrait, dont voici un exemple :

```
<xs:complexType name="mon_type">
 <xs:attribute name="att1" type="xs:string"/>
</xs:complexType>
<xs:complexType name="mon_type_herite_1">
 <complexContent>
  <extension base="mon_type">
   <xs:attribute name="att2" type="xs:integer"/>
  </extension>
 </complexContent>
</xs:complexType>
<xs:complexType name="mon_type_herite_2">
 <complexContent>
  <extension base="mon_type">
```

```
    <xs:attribute name="att3" type="xs:boolean"/>
   </extension>
  </complexContent>
</xs:complexType>
```

Dans un document, tout élément possédant le type `mon_type` pourra utiliser n'importe quel type hérité, c'est-à-dire `mon_type_herite_1` ou `mon_type_herite_2`.

Les conséquences de ces différences sont les suivantes :

- Le fait que les schémas soient basés sur les types implique que le contenu et les attributs d'un élément ne sont plus fixés (uniques), comme cela était le cas avec une DTD. Selon le contexte, un élément aura différents types. Il n'est donc plus possible de mettre directement en relation un élément et une table prédéfinie du modèle relationnel.
- La notion d'héritage augmente la complexité : étant donné un document XML, on ne peut pas prévoir le type de certains de ses éléments. Cela se produit lorsqu'un élément déclare un type qui possède plusieurs types hérités. Le type sera alors découvert lors de l'analyse du contenu du document XML grâce à l'attribut `xsi:type` qui permet à l'auteur du document de déclarer pour un élément le type qu'il a choisi, levant ainsi toute ambiguïté.

En conclusion de ce que nous venons de voir, nous pouvons retenir que XML Schema place *de facto* le problème de mise en correspondance des données XML avec des tables relationnelles entre la première et la deuxième des approches étudiées dans le début de ce chapitre.

XML Schema requiert toutefois que l'on regarde de plus près ce qui se passe pour chaque cas d'unité d'information : les éléments et leurs attributs, la structure, le contenu textuel.

Les éléments

Avec XML Schema, le type d'un élément peut fluctuer, y compris à l'intérieur même du document XML, quand on utilise le typage flottant autorisé avec l'attribut `xsi:type`. Les formes possibles d'un ensemble d'information rattaché à un schéma XML peuvent considérablement varier, contrairement aux DTD. Nous avons vu qu'il n'est pas possible d'assigner un élément à une table finie comme dans l'approche DTD. Cette dernière approche fonctionne car le nombre de colonnes, qui dépend du nombre d'attributs, est connu à l'avance, et parce que ce nombre est fixe. Cependant, dans le cas des schémas, ce nombre varie avec la possibilité de donner, à la volée, un type à un élément. Cet héritage dynamique de type est illustré par les figures 4-10 et 4-11.

La solution que nous allons développer fait reposer la mise en correspondance du XML avec le relationnel sur la base des types et non plus des éléments. Autrement

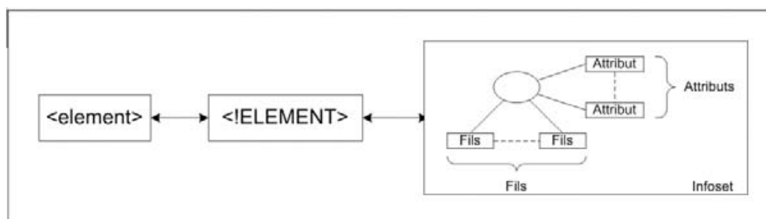


Figure 4–10
La relation biunivoque entre éléments et tables de l'approche DTD

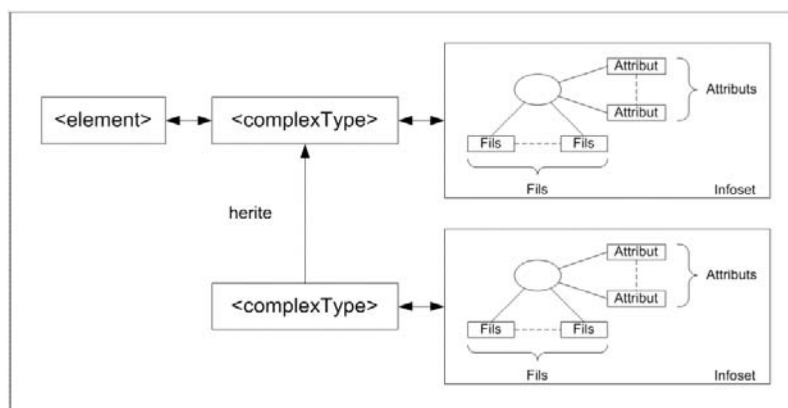


Figure 4–11
La relation entre éléments et tables des modèles par type (XML Schema)

dit, chaque élément d'un document XML sera stocké dans une table correspondant à son type et non plus à son nom d'élément.

Nous présentons tout d'abord la persistance de la structure arborescente et montrons que, sur ce point, les deux méthodes vues précédemment peuvent être utilisées. Ensuite, nous verrons comment prendre en compte les éléments textuels de manière efficace.

La structure arborescente

Les figures 4-12 et 4-13 décrivent les modèles relationnels qu'il convient d'appliquer selon que l'on souhaite suivre une approche DOM ou une approche DTD.

Si l'option DOM est choisie (figure 4-12), la table `élément` décrit la structure hiérarchique du document :

- `ID` : ce champ contient la clé primaire de l'élément.
- `TABLE` : ce champ contient le nom de la table dans laquelle l'élément est stocké.
- `NODE_INDEX` : ce champ sert à retenir l'ordre des fils d'un élément.

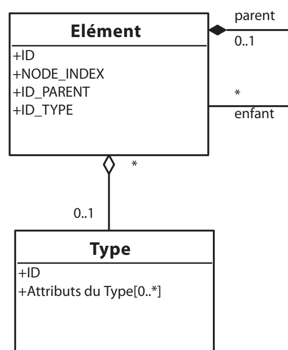


Figure 4–12
Diagramme UML de la méthode DOM appliquée à l'approche XML Schema

- `PARENT_ID` : ce champ, clé étrangère sur le champ `ID` de cette même table, donne pour un élément la clé de son père.

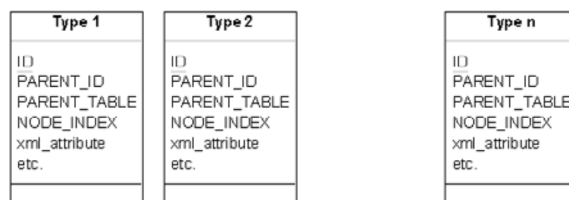


Figure 4–13
Diagramme UML de la méthode DTD appliquée à l'approche XML Schema

Si l'option DTD est retenue (figure 4-13), les relations parent-enfant sont distribuées sur les tables contenant les éléments.

La comparaison entre les deux modèles obtenus apporte les conclusions suivantes :

- Pour un élément donné, trouver l'ensemble de ses enfants est simple avec l'approche DOM et compliqué avec l'approche DTD. En effet, avec cette dernière, il faut itérer la recherche sur toutes les tables contenant des fils potentiels et rassembler tous les résultats. Cela a bien évidemment un coût en termes de performances tandis qu'avec l'approche DOM, le nombre de requêtes nécessaires sera réduit au strict minimum.
- Obtenir le père d'un élément donné : cette opération est similaire dans les deux cas.

On voit donc qu'il est préférable de choisir l'approche DOM.

> Lorsqu'un élément est stocké dans la table correspondant à son type, il faut conserver le nom que l'élément porte dans le document XML. On peut procéder de deux façons différentes. Soit on adjoint un champ dans la table `élément` (figure 4-12) dans lequel on viendra stocker le nom de l'élément, soit on ajoute ce champ sur chaque table (`Type 1`, `Type n`…, des figures 4-12 et 4-13) qui représente un type.

Les éléments textuels

Le stockage des éléments textuels est effectué dans une table spéciale qui contient les contenus de tous les éléments textuels du document :

Figure 4–14
Table de stockage
des données textuelles

Text
ID
DATA

Le stockage du texte mixte nécessite un traitement spécifique.

Stockage dans une base de données XML

En comparaison des problèmes que pose la projection du modèle relationnel, le principe du stockage du XML dans une base native XML est simple.

On qualifie de native une base de données dont la gestion des index repose sur le principe des arbres et des relations parents-enfants. Il existe une algèbre de parcours des arbres XML, tout comme il existe une algèbre du modèle relationnel. Ainsi, les documents XML ne sont pas transformés au moment de leur stockage : on ne cherche pas à faire rentrer une roue dans un carré et il n'y a par conséquent aucune distorsion.

Les nœuds de l'arbre correspondant à un document XML peuvent être identifiés par des *uplets* `(i,j)`.

Dans le tableau 4-2, nous représentons un fragment simple ainsi que les valeurs `i,j`, correspondant aux éléments ou nœuds. Le calcul est simple : le document XML est représenté sous sa forme normale ASCII et, au fur et à mesure de sa lecture, un compteur est incrémenté par pas de 1 à chaque fois qu'un nouvel élément ouvrant ou fermant est rencontré.

Document XML	**INDEX**	
	I	**J**
<piloteWeb xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schémaLogiqueDePiloteWeb.xsd">	1	
<sites>	2	
<site unique-id="p1">	3	
<nom>pilote 1</nom>	4	5
<position x="3" y="3"/>	6	7
<photo nom="http://www.piloteWeb.com/p1.gif" format="gif"/>	8	9

Document XML	INDEX	
	I	J
`</site>`		10
`<site unique-id="p2">`	11	
`<nom>pilote 2</nom>`	12	13
`<position x="3" y="4"/>`	14	15
`<photo nom="http://www.piloteWeb.com/p2.png" format="png"/>`	16	17
`</site>`		18
`</sites>`		19
`</piloteWeb>`		20

Si l'on représente ce fragment sous forme d'arbre, on obtient la représentation de la figure 4-15.

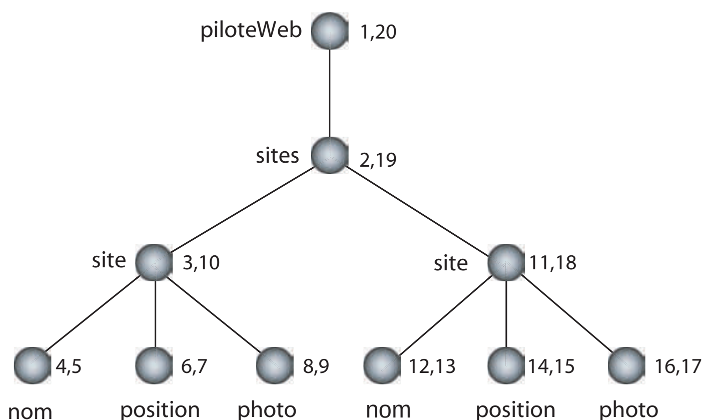


Figure 4–15
Indexation des éléments d'un document XML

Les opérations simples de navigation entre nœuds peuvent dès lors être calculées de la façon suivante :

- Le nœud frère d'un nœud de coordonnées `(i,j)` est forcément le nœud de coordonnées `(j+1,k)`.
- Le premier enfant d'un nœud de coordonnées `(i,j)` est le nœud de coordonnées `(i+1,k)`.
- Etc.

D'autres approches existent, mais notre objectif n'est pas dans cet ouvrage de les détailler toutes.

Une base de données XML se présente comme une arborescence de répertoires et de fichiers mais, à la différence du simple système de fichiers, elle offre des fonctions avancées d'indexation, requêtes, gestion des droits d'accès et contrôle des transactions.

Une base de données XML présente les caractéristiques suivantes :

- Les documents XML peuvent y être stockés même si le schéma ou la DTD qui les accompagnent est inconnu(e). Grâce aux algorithmes d'indexation et à la forme normalisée du XML, de nombreuses opérations de traitement (principalement, le requêtage et les modifications) sont possibles.
- Plusieurs documents XML peuvent cohabiter même si leurs schémas (ou DTD) sont différents.
- La gestion des révisions est prise en charge et le stockage des seules différences entre deux versions d'un même document permet d'optimiser la taille de la base.
- Il est aisé d'exporter tout ou partie de la base et de copier-coller tout ou partie d'un document XML entre la base et une autre application.

Ces types de système de gestion de bases de données combinent donc la commodité du système de fichiers avec les possibilités d'indexation et de contrôle des mises à jour.

La figure 4-16 montre une base de données XML.

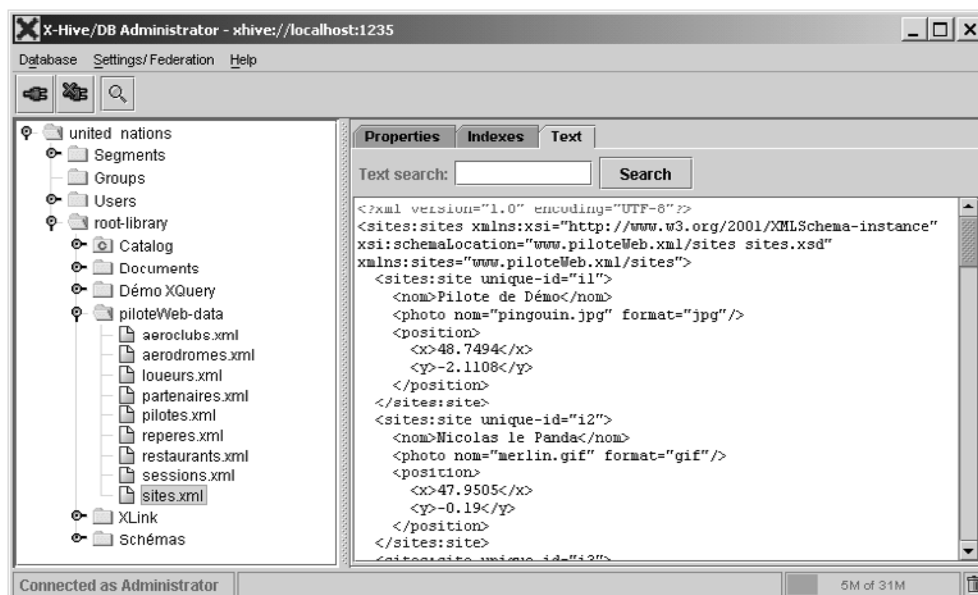


Figure 4–16 Dans une base de données XML, on navigue dans les données XML comme on parcourt un système de fichiers.

Toutes les opérations possibles sur les bases de données classiques le sont également avec une base de données XML, y compris les jointures de documents XML qui s'expriment, en XQuery, de la manière suivante :

```
for $i in document("/piloteWeb-data")//nom,
    $j in document("/Documents")//nomNorm
where $i = $j
return <res>{$i}</res>
```

Ici, la clause `for` permet de construire deux listes d'objets dont on demande ensuite le recoupement au moyen d'une clause `where`.

En résumé, le choix d'un stockage du XML dans une base relationnelle ou une base XML ne peut se discuter que :

- Par rapport à l'entropie que génère le stockage du XML dans le relationnel (le mot entropie représente ici l'énergie qu'il faut développer pour que cela marche – tenant compte des dégradations fonctionnelles que le relationnel apporte au XML).
- Par la nécessaire compatibilité avec d'autres applications : il se pourrait que les données XML ne fassent que représenter une partie des données d'une application plus vaste. Dans ce cas, c'est cette dernière qui décidera du mode de stockage.

Construire la stratégie d'adressage

Une fois le type de stockage défini, il faut aborder la question du mode d'adressage des données. XML offre plusieurs possibilités, toutes très précises :

- Le référencement par des identifiants uniques de type ID/IDREF : le problème, c'est que XML limite la portée de ce mécanisme aux liens intra-documents. L'utilisation des ID/IDREF ne marche qu'à l'intérieur d'un seul et même document XML. Il ne peut donc en aucun cas être utilisé pour répertorier (référencer) des données se trouvant dans plusieurs documents XML.
- Les URL permettent de cibler des éléments précis, d'un document XML précis, se trouvant sur une machine précise. Si les URL sont choisies, cela signifie que le stockage à l'intérieur de la base de données reproduit le système des fichiers : seule une base de données XML permet de le faire.
- Les URI nécessitent qu'un catalogue de correspondance entre des adresses physiques et logiques soit opérationnel.
- Les liens simples XLink : votre système de stockage doit appliquer cette recommandation, comme c'est généralement le cas.

- Les liens complexes XLink : contrairement aux liens simples, les liens XLink de type complexe sont rarement pris en charge.
- Les liens d'inclusion XInclude.
- Les expressions XPath qui permettent de définir un élément cible relativement à d'autres, voire de calculer une trajectoire. XPath reste un langage déclaratif dont les capacités de calcul sont limitées.
- Les requêtes XQuery reprennent le principe des expressions XPath en les enrobant d'un langage procédural. C'est un langage puissant qui permet d'adresser et créer artificiellement des liens entre des données qui n'auraient pas été initialement modélisées pour (des éléments porteurs d'aucun identifiant particulier par exemple).

À ce stade de la conception, vous devez faire la distinction entre :

- les identifiants, liens et références qui devront être posés manuellement par des auteurs des documents XML ;
- les identifiants, liens et références qui pourront être posés par des programmes ;
- les liens qui seront le fruit d'un calcul fait à la volée, en général au moment de l'affichage des données.

Dans le cas où l'information serait posée manuellement, la question est toujours de savoir si l'utilisateur dispose de l'information pour poser son lien. Question qui se décline ainsi : peut-il voir la cible ? Doit-il en connaître (et comment) les identifiants internes ? Peut-il poser son lien par une opération de copier-coller ? Etc.

Dans le cas du lien calculé par programme, la question posée est toujours celle de la mise à jour du lien quand l'information ciblée évolue : le lien est-il stable dans le temps ? Comment le mettre éventuellement à jour ?

Cas d'école : PiloteWeb

Dans cette section, nous présentons ce que serait le stockage des données de PiloteWeb dans chacun des trois cas de figure : le système de fichiers, des tables relationnelles, et enfin la base de données XML.

Découpage initial du modèle logique

Au chapitre 3, nous avions obtenu le modèle logique des données de PiloteWeb, avec une mise en évidence de la structure logique que nous représentons à nouveau ci-après à la figure 4-17.

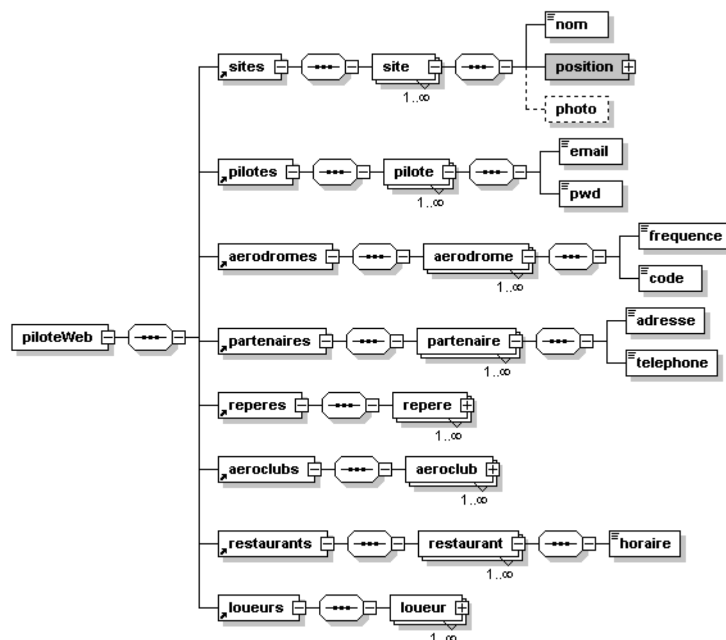


Figure 4–17
Représentation de la structure XML de PiloteWeb

L'analyse fonctionnelle de l'application fait apparaître que les mises à jour des données de la base sont assurées par plusieurs personnes, travaillant de manière synchrone sur la base :

- Les aéroclubs vont inscrire de nouveaux pilotes.
- Les autorités de régulation de la circulation aérienne vont déclarer de nouveaux aérodromes ou modifier les caractéristiques des aérodromes et aéroclubs existants.
- Les services des chambres de commerce régionales vont prendre en charge l'inscription des partenaires des aérodromes et aéroclubs.

Si nous conservons toutes les données dans un seul document XML, tel que notre modèle logique l'impose dans son état actuel, ces utilisateurs ne pourraient modifier les données qu'en procédant l'un après l'autre.

Pour réduire les temps d'accès à la base, nous allons donc décider de découper la structure physique en plusieurs parties, chacune pouvant être mise à jour indépendamment des autres.

Pour cela, nous allons transformer chacun des éléments enfants de l'élément `piloteWeb` en élément racine d'un document XML. Notre base sera ainsi composée de huit documents XML, correspondant à huit schémas différents. Nous optons donc pour un choix de base de type `n,1:n` schémas et un seul document XML par schéma.

La figure 4-18 donne la représentation du schéma du document XML qui contiendra les données relatives aux pilotes (élément `pilote`). Un nouvel élément `pilotes` (avec un s) a été introduit pour servir de racine à ce document XML.

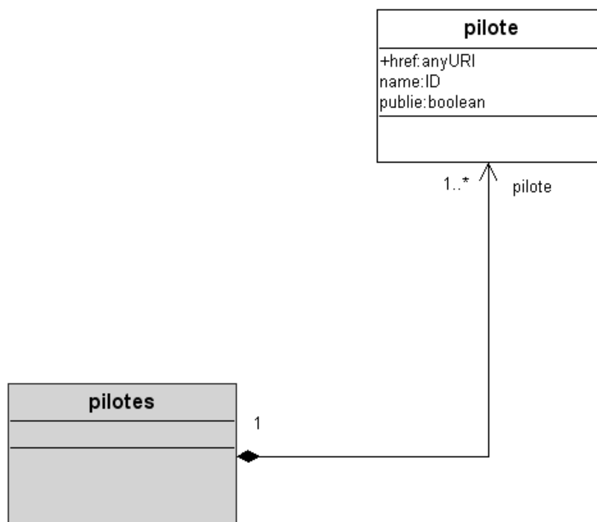


Figure 4–18
Représentation en UML du schéma correspondant au stockage physique des données relatives aux pilotes

Ci-après, nous présentons huit extraits des documents XML que nous allons obtenir suite à ce découpage.

> **REMARQUE anyURL**
>
> Le type `anyURL` n'existe pas dans la recommandation XMLSchema. Le type le plus proche que nous puissions utiliser est `anyURI`; ce que nous avons fait ici.

Un document XML pour les sites

```
<?xml version="1.0" encoding="UTF-8"?>
<sites xmlns:xsi="…" xsi:noNamespaceSchemaLocation="sites.xsd">
 <site unique-id="p1">
  <nom>Nicolas le Panda</nom>
  <position x="48.7494" y="-2.1108"/>
  <photo nom="http://www.piloteWeb.com/p1.gif" format="gif"/>
 </site>
</sites>
```

Un document XML pour les pilotes

```
<?xml version="1.0" encoding="UTF-8"?>
<pilotes>
 <pilote unique-id="id11" publie="1 idrefSite="p1">
  <email>pilote@piloteWeb.com</email>
  <pwd>avion</pwd>
 </pilote>
</pilotes>
```

Un document XML pour les aérodromes

```
<?xml version="1.0" encoding="UTF-8"?>
<aerodromes>
 <aerodrome unique-id="id13" idrefSite="a1">
  <frequence>125.90</frequence>
  <code>LFRM</code>
 </aerodrome>
</aerodromes>
```

Un document XML pour les partenaires

```
<?xml version="1.0" encoding="UTF-8"?>
<partenaires>
 <partenaire unique-id="idpa15" idrefSite="pa1">
  <adresse>ZI des Arbihres Saint-Nazaire </adresse>
  <telephone>+33.2.35.60.42.89</telephone>
 </partenaire>
</partenaires>
```

Un document XML pour les repères

```
<?xml version="1.0" encoding="UTF-8"?>
<reperes>
 <repere unique-id="id23" idrefSite="r1"/>
 <repere unique-id="id24" idrefSite="r2"/>
</reperes>
```

Un document XML pour les aéroclubs

```
<?xml version="1.0" encoding="UTF-8"?>
<aeroclubs>
 <aeroclub unique-id="id17" idrefPart="idpa15"/>
 <aeroclub unique-id="id18" idrefPart="idpa16"/>
</aeroclubs>
```

Un document XML pour les restaurants

```
<?xml version="1.0" encoding="UTF-8"?>
<restaurants>
 <restaurant unique-id="id19" idrefPart="idre15">
  <horaires>Dès 7 heures et jusqu’à 20h. tous les jours sauf le WE
  </horaires>
 </restaurant>
</restaurants>
```

Un document XML pour les loueurs

```
<?xml version="1.0" encoding="UTF-8"?>
<loueurs>
 <loueur unique-id="id21" idrefPart="idlo15"/>
 <loueur unique-id="id22" idrefPart="idlo16"/>
</loueurs>
```

Remarque : nous avons remplacé par … l’URI de XML Schema, à savoir `http://www.w3.org/2001/XMLSchema-instance`, et ce par souci d’économie de place.

Or, comme les données sont découpées dans plusieurs documents XML indépendants, le mécanisme des ID/IDREF initialement mis en place pour partager les données entre elles ne fonctionne plus. Par exemple, le fait d’écrire `<restaurant unique-id="id19" idrefPart="idre15">` ne suffit plus pour savoir où se trouve la donnée référencée puisqu’il manque désormais l’indication du fichier de la cible de cette référence. En l’état actuel, seule la programmation des chemins d’accès en dur dans les programmes de l’application permettrait de résoudre ce point.

Stockage dans un système de fichiers

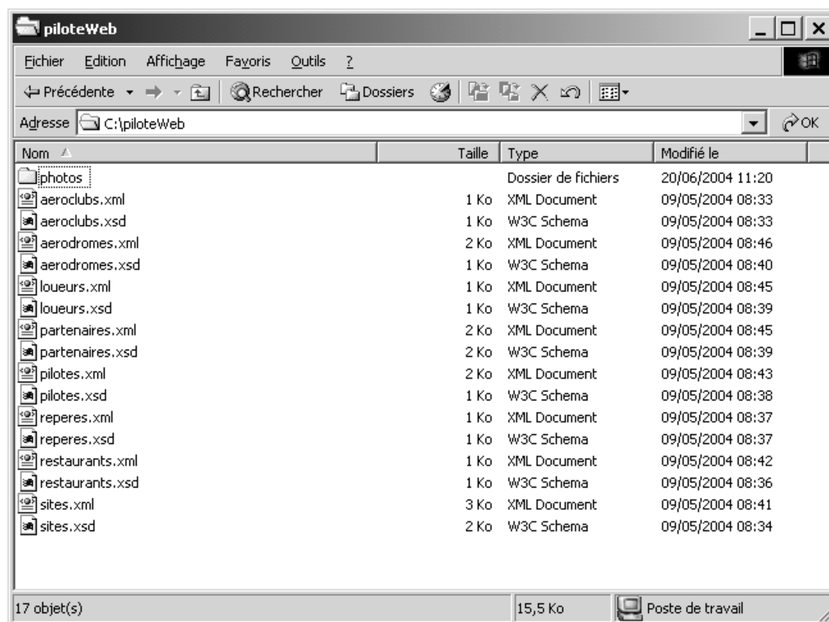
Nous allons prendre le cas simple où tous les documents XML et leurs schémas associés sont stockés dans un même répertoire.

À la figure 4-19, nous représentons un répertoire dans lequel sont stockés indifféremment les schémas XML et les documents XML correspondants.

À la manière des documents HTML, nous utilisons dans ce cas des URL pour exprimer les liens entre les données, et ce afin d’éviter à l’application de devoir gérer cette partie de la logique du modèle de données.

Il faut aussi prévoir un lieu de stockage pour les éléments graphiques de l’application : les images qui correspondent à l’élément `photo` doivent se trouver dans un répertoire connu de l’application.

Figure 4–19
Arborescence physique de PiloteWeb dans le cas d'un stockage dans un système de fichiers



Nous montrons ci-dessous ce que cela donnerait dans les documents de PiloteWeb.

Pour garantir la qualité de l'environnement, il conviendra de modifier les schémas de ces documents XML pour y adapter tant les noms des attributs que leur type.

Nous allons montrer dans les sections suivantes comment les schémas XML, logique et conceptuel, relatifs à l'élément `pilotes` s'en trouvent modifiés.

Un document XML pour les sites (`Unique-id` remplacé par `name`)

```
<?xml version="1.0" encoding="UTF-8"?>
<sites xmlns:xsi="…" xsi:noNamespaceSchemaLocation="sites.xsd">
 <site name="p1">
  <nom>Nicolas le Panda</nom>
  <position x="48.7494" y="-2.1108"/>
  <photo nom="photos/p1.gif"
         format="gif"/>
 </site>
</sites>
```

Un document XML pour les pilotes (`IdrefSite` remplacé par `href`)

```
<?xml version="1.0" encoding="UTF-8"?>
<pilotes>
 <pilote name="id11" publie="1" href="sites.xml#p1">
  <email>pilote@piloteWeb.com</email>
  <pwd>avion</pwd>
 </pilote>
</pilotes>
```

Un document XML pour les aérodromes (`Unique-id` remplacé par `name`, `IdrefSite` remplacé par `href`)

```
<?xml version="1.0" encoding="UTF-8"?>
<aerodromes>
 <aerodrome name="id13" href="sites.xml#a1">
  <frequence>125.90</frequence>
  <code>LFRM</code>
 </aerodrome>
</aerodromes>
```

Un document XML pour les partenaires (`Unique-id` remplacé par `name`, `IdrefSite` remplacé par `href`)

```
<?xml version="1.0" encoding="UTF-8"?>
<partenaires>
 <partenaire name="idpa15" href="sites.xml#pa1">
  <adresse>ZI des Arbihres Saint-Nazaire</adresse>
  <telephone>+33.2.35.60.42.89
  </telephone>
 </partenaire>
</partenaires>
```

Un document XML pour les repères (`Unique-id` remplacé par `name`, `IdrefSite` remplacé par `href`)

```
<?xml version="1.0" encoding="UTF-8"?>
<reperes>
 <repere name="id23" href="sites.xml#r1"/>
 <repere name="id24" href=" sites.xml#r2"/>
</reperes>
```

Un document XML pour les aéroclubs (`Unique-id` remplacé par `name`, `IdrefSite` remplacé par `href`)

```
<?xml version="1.0" encoding="UTF-8"?>
<aeroclubs>
 <aeroclub name="id17" href="partenaires.xml#idpa15"/>
 <aeroclub name="id18" href="partenaires.xml#idpa16"/>
</aeroclubs>
```

Un document XML pour les restaurants (`Unique-id` remplacé par `name`, `IdrefSite` remplacé par `href`)

```
<?xml version="1.0" encoding="UTF-8"?>
<restaurants>
 <restaurant name="id19" href="partenaires.xml#idre15">
  <horaires>Dès 7 heures et jusqu'à 20h. tous les jours sauf le WE
  </horaires>
 </restaurant>
</restaurants>
```

Un document XML pour les loueurs (`Unique-id` remplacé par `name`, `IdrefSite` remplacé par `href`)

```
<?xml version="1.0" encoding="UTF-8"?>
<loueurs>
 <loueur name="id21" href="partenaires.xml#idlo15"/>
 <loueur name="id22" href="partenaires.xml#idlo16"/>
</loueurs>
```

Stockage en relationnel

Dans cette section, nous allons présenter la façon dont les modèles et documents de PiloteWeb ont pu être stockés dans le produit Oracle XML DB. (Oracle 9iR2).

Comme nous allons le voir, le contenu des documents XML est à la fois réparti dans des tables et sous la forme d'une arborescence de fichiers. L'arborescence hiérarchique est constituée de nœuds, chacun d'eux pouvant être un document XML ou un dossier. La répartition des documents est identique à celle utilisée pour le stockage dans un système de fichiers. Vous pouvez choisir l'arborescence et les lieux de stockage des objets.

L'accès à ce référentiel est obtenu par les protocoles HTTP, FTP, WebDAV, ce que représente la figure 4-20.

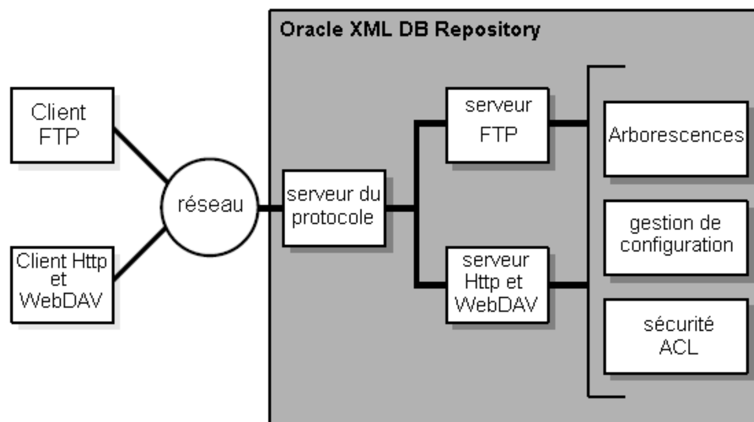


Figure 4–20
Architecture Oracle XML DB – Les protocoles disponibles

Avec ce produit, il faut distinguer :

1. les vues qui permettent de représenter une arborescence de fichiers à la manière d'un système de fichiers ;
2. les vues qui représentent les structures XML internes des documents stockés.

Vues représentant l'arborescence des fichiers

De par sa capacité à représenter et gérer les documents et schémas XML sous la forme d'une arborescence de fichiers, Oracle XML DB se présente comme un système de stockage hiérarchique de document XML. On peut accéder à ces documents en utilisant les protocoles HTTP, FTP ou WebDAV, ou encore en utilisant l'un des langages de requêtes SQL, PLSQL ou Java.

Pour stocker les schémas dans la base, la première opération consiste à charger les fichiers contenant les schémas XML dans l'un des répertoires virtuels déjà créés dans la base. En effet, Oracle XML DB permet de créer de tels répertoires. Nous montrons ci-après un exemple de scripts de création de tels répertoires :

```
set echo on
connect &1/&2@&3
--
declare
  result boolean;
begin
  result := dbms_xdb.createFolder('/home/' || USER || '/xsd');
```

```
  result := dbms_xdb.createFolder('/home/' || USER || '/xsl');
  result := dbms_xdb.createFolder('/home/' || USER || '/PiloteWeb');
result := dbms_xdb.createFolder('/home/' || USER || '/photos');
end;
/
COMMIT;
```

Ce script donne une arborescence que l'on peut consulter à partir d'un navigateur Web (résultat illustré par la figure 4-21).

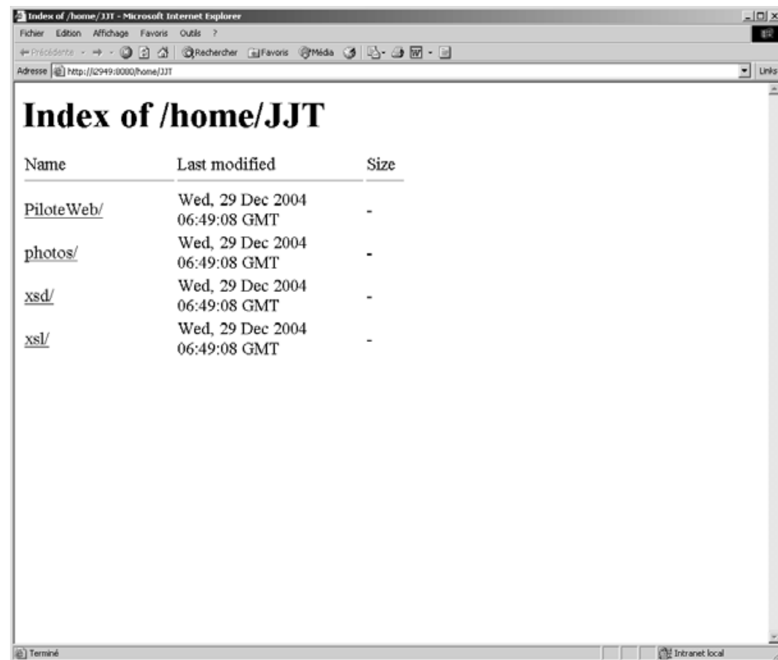


Figure 4–21
Création des répertoires de PiloteWeb dans Oracle XML DB

C'est dans le répertoire `/home/JJT/xsd` que seront stockés les schémas XML. On peut y procéder au moyen d'un téléchargement via FTP.

Les figures 4-22, 4-23 et 4-24 montrent les différentes étapes de ce téléchargement, soit respectivement :

- la connexion à la base ;
- la sélection d'un répertoire de destination ;
- le dépôt des schémas.

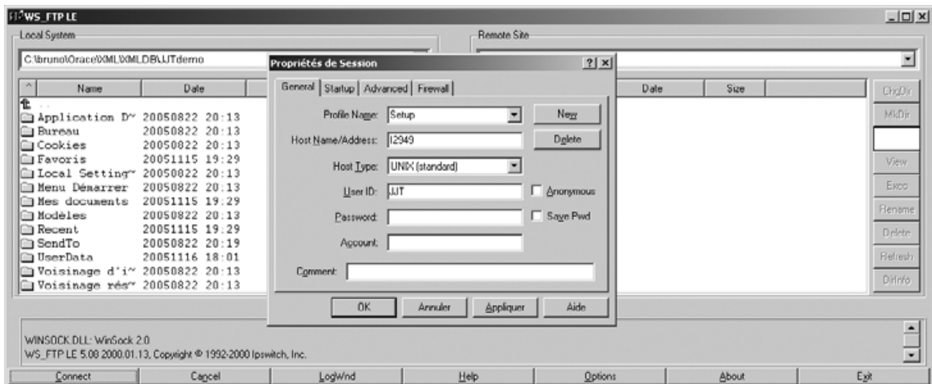


Figure 4–22 Connexion à Oracle XML DB

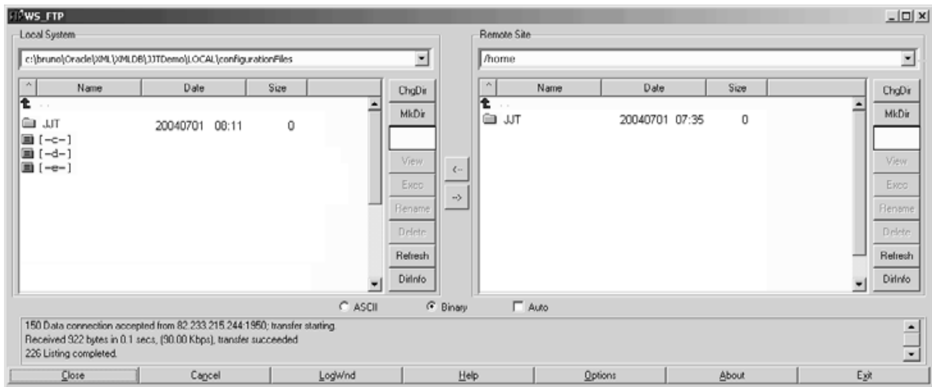


Figure 4–23 Sélection du répertoire virtuel de destination

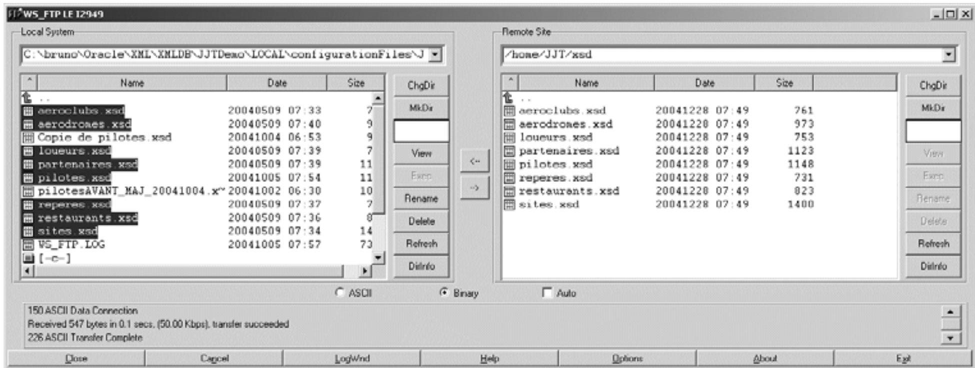


Figure 4–24 Dépôt des schémas dans le répertoire virtuel de destination

Oracle XML DB charge les informations des fichiers déposés dans deux vues principales :

- La vue `RESOURCE_VIEW` : elle contient une ligne par ressource enregistrée dans la base.
- La vue `PATH_VIEW` : elle contient les chemins d'accès aux ressources stockées dans la base.

La figure 4-25 représente ces deux vues.

```
SQL> desc resource_view
Name                           Nul?    Type
------------------------------ ------- ----------------
RES                                    SYS.XMLTYPE
ANY_PATH                               VARCHAR2(4000)

SQL> desc path_view
Name                           Nul?    Type
------------------------------ ------- ----------------
PATH                                   VARCHAR2(1024)
RES                                    SYS.XMLTYPE
LINK                                   SYS.XMLTYPE

SQL>
```

Figure 4–25
Les vues resource et path de Oracle XML DB

La figure 4-26 montre la requête SQL à exécuter pour avoir la liste des chemins d'accès aux ressources stockées dans la base, tandis que la figure 4-27 montre la requête fournissant, pour l'un des schémas stockés, les informations détaillées le concernant, et qui se trouvent dans la table `resource_view`.

```
SQL> select any_path from resource_view where any_path like '/home/JJT/xsd/%' ;

ANY_PATH
-------------------------------------------------------
/home/JJT/xsd/aeroclubs.xsd
/home/JJT/xsd/aerodromes.xsd
/home/JJT/xsd/loueurs.xsd
/home/JJT/xsd/partenaires.xsd
/home/JJT/xsd/pilotes.xsd
/home/JJT/xsd/reperes.xsd
/home/JJT/xsd/restaurants.xsd
/home/JJT/xsd/sites.xsd

8 rows selected.

SQL>
```

Figure 4–26
Requête SQL montrant les schémas identifiés dans la table resource_view

Comme nous l'avons déjà fait, il est possible, via HTTP, d'interroger directement la base en utilisant simplement l'URL http://i2949:8080/home/JJT/xsd, comme le montre la figure 4-28.

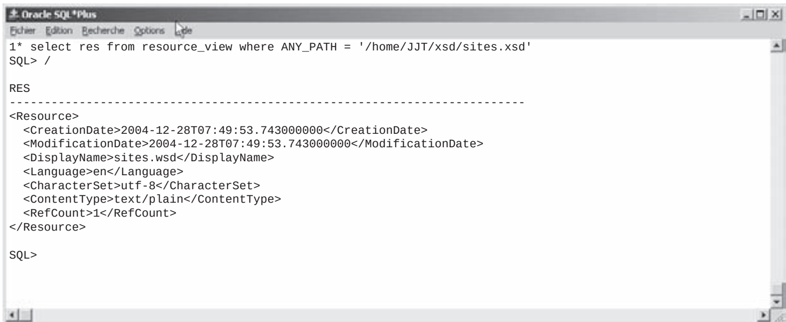


Figure 4–27
Requête permettant d'obtenir les informations de gestion pour l'un des schémas stockés

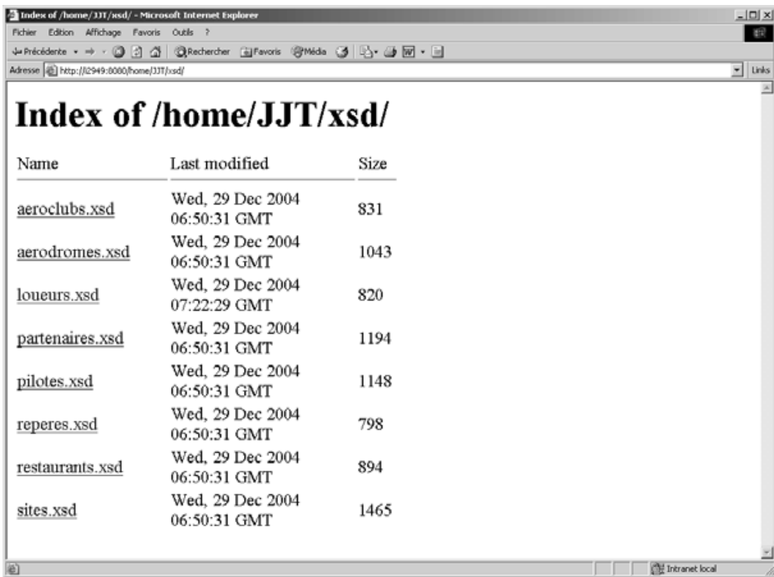


Figure 4–28
Requête HTTP fournissant les données de gestion des schémas

En cliquant sur l'un des noms de schémas, le contenu s'affiche dans le navigateur aussi simplement que s'il s'agissait d'un fichier stocké dans un répertoire du système d'exploitation. Dans la figure 4-29, vous voyez le résultat après avoir cliqué sur `aeroclubs.xsd`.

Dans cette vue, vous pouvez immédiatement remarquer l'utilisation de l'attribut `xdb:defaultTable="AEROCLUBS"`, dont la présence montre que le schéma XML a dû être adapté avant son chargement dans la base. Ce point sera étudié dans la prochaine section.

La dernière vue possible est celle obtenue via WebDAV, dans laquelle les répertoires virtuels de la base apparaissent comme autant de répertoires du système d'exploitation (figure 4-30).

Figure 4–29
Affichage dans le navigateur d'un des schémas stockés dans Oracle XML DB

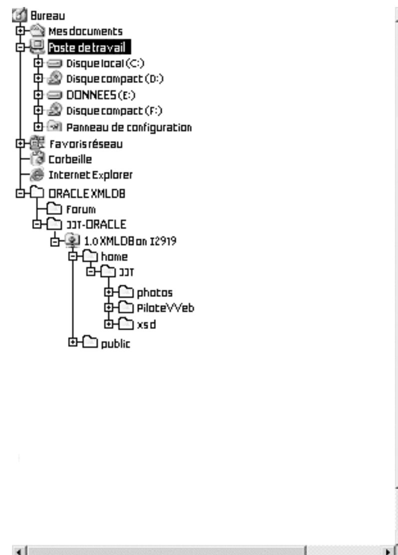


Figure 4–30
Accès au répertoire virtuel de la base à partir du Bureau Windows

Vues relatives aux schémas

Dans cette section, nous allons voir comment Oracle XML DB gère la structure des schémas XML que nous lui avons demandé de stocker.

En observant la figure 4-29, il apparaît que deux informations ont été ajoutées dans le schéma XML :

- une déclaration d'espace de noms, à savoir : `xmlns:xdb="http://xmlns.oracle.com/xdb"` ;
- un attribut spécifiant un nom de table, à savoir `xdb:defaultTable="AEROCLUBS"`.

De telles informations ainsi que les scripts présentés dans cette section vont permettre de prendre en compte la structure des schémas XML :

- génération de la structure en fonction du schéma XML ;
- création des tables, vues et colonnes XMLTYPE à partir de schémas XML enregistrés.

On pourra ensuite effectuer le stockage automatique des documents dans les tables qui correspondent à un ou plusieurs schéma(s) XML enregistré(s).

Pour que la structure définie par le schéma soit enregistrée dans la base, il faut demander au système de le faire. Nous montrons ci-après les commandes utilisées à cet effet.

```
Set echo on
connect &1/&2@&3
--
alter session Set events='31098 trace name context forever' ;
begin
dbms_xmlschema.registerSchema(
   'http://I2949:8080/JJT/sites.xsd',
   xdbURIType('/home/JJT/xsd/sites.xsd').getClob(),
   TRUE,TRUE,FALSE,TRUE);
dbms_xmlschema.registerSchema(
   'http://I2949:8080/home/JJT/xsd/aeroclubs.xsd',
   xdbURIType('/home/JJT/xsd/aeroclubs.xsd').getClob(),
   TRUE,TRUE,FALSE,TRUE);
dbms_xmlschema.registerSchema(
   'http://I2949:8080/home/JJT/xsd/aerodromes.xsd',
   xdbURIType('/home/JJT/xsd/aerodromes.xsd').getClob(),
   TRUE,TRUE,FALSE,TRUE);
dbms_xmlschema.registerSchema(
   'http://I2949:8080/home/JJT/xsd/loueurs.xsd',
   xdbURIType('/home/JJT/xsd/loueurs.xsd').getClob(),
   TRUE,TRUE,FALSE,TRUE);
```

```
dbms_xmlschema.registerSchema(
   'http://I2949:8080/home/JJT/xsd/partenaires.xsd',
   xdbURIType('/home/JJT/xsd/partenaires.xsd').getClob(),
   TRUE,TRUE,FALSE,TRUE);
dbms_xmlschema.registerSchema(
   'http://I2949:8080/home/JJT/xsd/pilotes.xsd',
   xdbURIType('/home/JJT/xsd/pilotes.xsd').getClob(),
   TRUE,TRUE,FALSE,TRUE);
dbms_xmlschema.registerSchema(
   'http://I2949:8080/home/JJT/xsd/reperes.xsd',
   xdbURIType('/home/JJT/xsd/reperes.xsd').getClob(),
   TRUE,TRUE,FALSE,TRUE);
dbms_xmlschema.registerSchema(
   'http://I2949:8080/home/JJT/xsd/restaurants.xsd',
   xdbURIType('/home/JJT/xsd/restaurants.xsd').getClob(),
   TRUE,TRUE,FALSE,TRUE);
End;
/
```

À la figure 4-31, nous montrons la liste des tables résultant de cette opération d'enregistrement.

Figure 4–31
Liste des tables générées à partir des schémas XML du référentiel

```
Oracle SQL*Plus
File Edit Search Options Help
SQL> select object_name, object_type from user_objects
  2  where object_type='TABLE';

OBJECT_NAME                              OBJECT_TYPE
---------------------------------------- ------------------
AEROCLUBS                                TABLE
AERODROMES                               TABLE
LOUEURS                                  TABLE
PARTENAIRES                              TABLE
PILOTE                                   TABLE
PILOTES                                  TABLE
REPERES                                  TABLE
RESTAURANTS                              TABLE
SITES                                    TABLE
aerodrome442_TAB                         TABLE
partenaire453_TAB                        TABLE
site431_TAB                              TABLE

12 rows selected.

SQL>
```

Nous allons analyser la décomposition d'un schéma XML en table. Pour illustrer cette analyse, nous allons utiliser les tables `pilotes` et `pilote` issues du schéma XML `pilotes.xsd`. À cet effet, nous avons dû apporter quelques modifications au schéma XML des pilotes.

Elles sont mises en valeur à la figure 4-32.

Figure 4–32
Le schéma pilotes.xsd soumis à Oracle XML DB

```
  <?xml version="1.0" encoding="iso-8859-1" ?>
- <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:xdb="http://xmlns.oracle.com/xdb" version="1.0">
 - <xs:complexType name="piloteType" xdb:SQLType="XDBPI_TYPE">
   - <xs:sequence>
     - <xs:element name="link">
       - <xs:complexType>
           <xs:anyAttribute
             namespace="http://www.w3.org/1999/xlink"
             processContents="strict" />
         </xs:complexType>
       </xs:element>
       <xs:element name="email" type="emailType" nillable="false"
        xdb:SQLName="MAIL" />
       <xs:element name="pwd" type="xs:NCName" nillable="false"
        xdb:SQLName="PWD" />
     </xs:sequence>
     <xs:attribute name="unique-id" type="xs:ID" />
     <xs:attribute name="publie" type="xs:integer" default="1" />
   </xs:complexType>
 - <xs:simpleType name="emailType">
   - <xs:restriction base="xs:string">
       <xs:pattern value="(\c)*@(\c)*" />
     </xs:restriction>
   </xs:simpleType>
   <xs:element name="pilote" type="piloteType"
    xdb:defaultTable="PILOTE" />
 - <xs:element name="pilotes" xdb:defaultTable="PILOTES">
   - <xs:complexType>
     - <xs:sequence>
         <xs:element ref="pilote" maxOccurs="unbounded" />
       </xs:sequence>
     </xs:complexType>
   </xs:element>
```

```
SQL> set long 1000
SQL> set wrap ON
SQL> set linesize 200
SQL> desc pilotes
TABLE of SYS.XMLTYPE(XMLSchema "http://I2949:8080/home/JJT/xsd/pilotes.xsd"
Element "pilotes") STORAGE Object-relational TYPE "pilotes457_T"

SQL>
SQL> desc "pilotes457_T"
"pilotes457_T" is NOT FINAL
```

Voici le résultat de cette requête :

Name	Type
SYS_XDBPD$	XDB.XDB$RAW_LIST_T
Pilote	pilote458_COLL

Dans notre schéma, l'élément `pilotes` contient un seul élément, pilote, défini globalement par un type complexe nommé ; ce dernier fait l'objet d'une table particulière (la table `pilote458_COLL`). Nous en demandons la structure via la requête suivante :

```
SQL> desc "pilote458_COLL"
"pilote458_COLL" VARRAY(2147483647) OF XDBPI_TYPE
"pilote458_COLL" is NOT FINAL
```

Voici le résultat de cette requête :

Name	Type
SYS_XDBPD$	XDB.XDB$RAW_LIST_T
unique-id	VARCHAR2(4000)
Publie	NUMBER(38)
Link	Link455_T
MAIL	VARCHAR2(4000)
PWD	VARCHAR2(4000)

La table du type `piloteType` (`pilote458_COLL`) stocke directement dans ses colonnes les objets de type simple, qu'il s'agisse des attributs (`unique-id`, `Publie`) ou des sous-éléments (`MAIL`, `PWD`) du type. On remarquera que les éléments `email` et `pwd` ont été stockés sous les noms `MAIL` et `PWD` puisque cela a été explicitement demandé dans le schéma (voir encadrés de la figure 4-32).

Le sous-élément `link`, qui est de type complexe, fait l'objet d'une table séparée. Nous en demandons la structure via la requête suivante :

```
SQL> desc "link455_T"
"link455_T" is NOT FINAL
```

Voici le résultat de cette requête :

Name	Type
SYS_XDBPD$	XDB.XDB$RAW_LIST_T
SYS_XDBANYATTR456$	VARCHAR2(4000)

Du fait du type générique (`anyAttribute`) qui sert à définir l'élément `link`, Oracle XML DB décide de créer un seul long champ.

L'élément `pilote` pointe vers la table de son type. Quand on demande sa structure, on obtient une description déjà rencontrée plus haut :

```
SQL> desc pilote
TABLE of SYS.XMLTYPE(XMLSchema "http://I2949:8080/home/JJT/xsd/pilotes.xsd"
Element "pilote") STORAGE Object-relational TYPE "XDBPI_TYPE"
```

Et le résultat de cette requête est le suivant :

```
SQL> desc XDB.XDB$RAW_LIST_T
XDB.XDB$RAW_LIST_T VARRAY(1000) OF RAW(2000)
```

Name	Type
SYS_XDBPD$	XDB.XDB$RAW_LIST_T
unique_id	VARCHAR2(4000)
Publie	NUMBER(38)
Link	Link455_T
MAIL	VARCHAR2(4000)
PWD	VARCHAR2(4000)

Finalement, la figure 4-33 représente la structure des tables créées par Oracle XML DB.

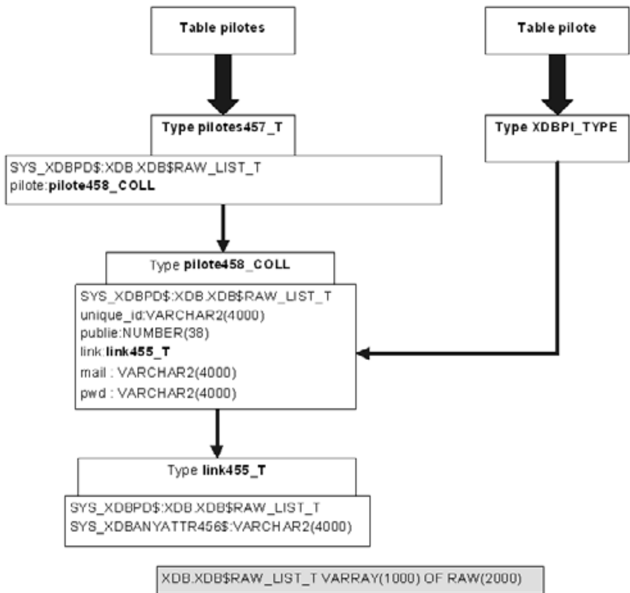


Figure 4–33
Représentation des tables créées par Oracle pour le schéma pilotes.xsd

La récupération des fichiers XML

Pour récupérer les fichiers, on peut également procéder de plusieurs manières : au moyen de FTP ou WebDAV.

Pour illustrer cette phase, nous allons vérifier le résultat de la récupération, avec le fichier XML `pilotes.xml`, chargé dans le répertoire `/home/JJT/PiloteWeb` (figure 4-34).

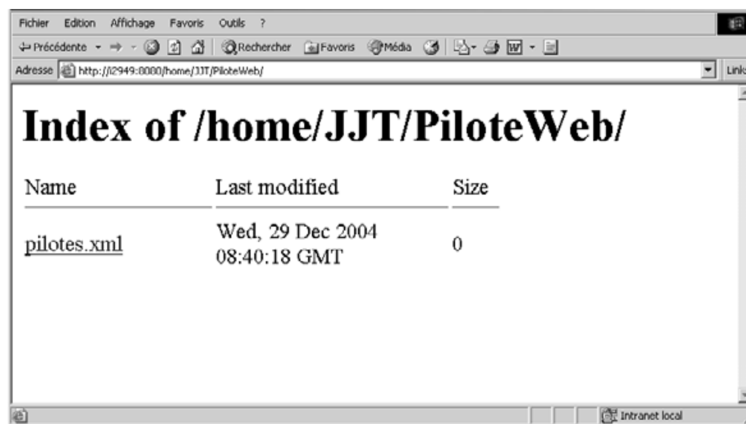


Figure 4–34
Contenu du référentiel après importation du fichier pilotes.xml par FTP

Cependant, avant de charger le fichier XML dans la base de données, il faut changer son en-tête pour y indiquer le chemin d'accès au schéma correspondant dans Oracle (repère ❶).

```
<pilotes xmlns:xlink=http://www.w3.org/1999/xlink
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xsi:noNamespaceSchemaLocation="http://I2949:8080/home/JJT/xsd/
➥ pilotes.xsd"> ← ❶
```

Après le chargement du fichier, on peut vérifier le contenu de la base Oracle en exécutant la requête SQL suivante :

```
SQL> select count(*) from PILOTES;
        1
```

Oracle n'a stocké qu'une ligne au niveau de la table `pilotes`, et cette ligne contient l'ensemble du fichier XML.

On peut toutefois afficher la ligne de la table `pilotes` dont un élément `email` contient la valeur `max@piloteweb.com`.

```
SQL> select value(x) from pilotes x
  2 where existsNode(value(x),'//pilote[email="max@piloteweb.com"]') = 1;
```

```
<pilotes xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://I2949:8080/home/JJT/xsd/pilotes.xsd">
  <pilote unique-id="i16" publie="1">
    <!--link xlink:type="simple" xlink:show="embed" xlink:href=
        "sites.xml/#xpointer(descendant::site[@unique-id='i1'])"/-->
    <email>pilote@piloteweb.com</email>
    <pwd>avion</pwd>
  </pilote>
  <pilote unique-id="i17" publie="1">
    <!--link xlink:type="simple" xlink:show="embed" xlink:href=
         "sites.xml/#xpointer(descendant::site[@unique-id='i2'])"/-->
    <email>panda@piloteweb.com</email>
    <pwd>secret</pwd>
  </pilote>
  <pilote unique-id="i18" publie="1">
    <!--link xlink:type="simple" xlink:show="embed" xlink:href=
         "sites.xml/#xpointer(descendant::site[@unique-id='i3'])"/-->
    <email>max@piloteweb.com</email>
    <pwd>secret</pwd>
  </pilote>
  <pilote unique-id="i19" publie="1">
    <!--link xlink:type="simple" xlink:show="embed" xlink:href=
         "sites.xml/#xpointer(descendant::site[@unique-id='i4'])"/-->
    <email>snail@piloteweb.com</email>
    <pwd>secret</pwd>
  </pilote>
</pilotes>
```

La requête SQL a trouvé dans la table un élément correspondant au critère de recherche. Comme la table ne contient qu'une ligne, cette requête ramène la ligne complète, et donc le contenu du fichier XML.

En revanche, si on utilise la requête suivante, on obtient en retour le seul élément `email` requis :

```
SQL> select x.extract('//pilote/email') from pilotes x where
existsNode(value(x),'//pilote[email="max@piloteweb.com"]') = 1;
<email>max@piloteweb.com</email>
```

Pour obtenir ce résultat, c'est la fonction spécifique `x.extract` qui a été utilisée au lieu du standard `select`.

La mise à jour d'un fichier XML

Il est possible d'utiliser des chemins XPath dans des requêtes SQL de mise à jour. En voici un exemple :

```
SQL> update pilotes p
  2 set value(p) =
  3 updateXml(value(p),'//pilote[3]/email/text()','maxi@piloteweb.com')
  4 where existsNode(value(p),
  5        '//pilote[position()= 3 and email="max@piloteweb.com"]') = 1
  6 /
```

Cette requête met à jour la chaîne de caractères de l'élément `email` pointé par l'expression XPath (repère ❶) :

```
1 row updated.
SQL> select xdbURIType('/home/' || USER
                     || '/PiloteWeb/pilotes.xml').getXML() from dual;

<pilotes xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://I2949:8080/home/JJT/xsd/pilotes.xsd">
  <pilote unique-id="i16" publie="1">
    <!--link xlink:type="simple" xlink:show="embed" xlink:href=
         "sites.xml/#xpointer(descendant::site[@unique-id='i1'])"/-->
    <email>pilote@piloteweb.com</email>
    <pwd>avion</pwd>
  </pilote>
  <pilote unique-id="i17" publie="1">
    <!--link xlink:type="simple" xlink:show="embed" xlink:href=
         "sites.xml/#xpointer(descendant::site[@unique-id='i2'])"/-->
    <email>panda@piloteweb.com</email>
    <pwd>secret</pwd>
  </pilote>
  <pilote unique-id="i18" publie="1">
    <!--link xlink:type="simple" xlink:show="embed" xlink:href=
         "sites.xml/#xpointer(descendant::site[@unique-id='i3'])"/-->
    <email>maxi@piloteweb.com</email> ← ❶
    <pwd>secret</pwd>
  </pilote>
  <pilote unique-id="i19" publie="1">
    <!--link xlink:type="simple" xlink:show="embed" xlink:href=
         "sites.xml/#xpointer(descendant::site[@unique-id='i4'])"/-->
    <email>snail@piloteweb.com</email>
    <pwd>secret</pwd>
  </pilote>
</pilotes>
```

Stockage dans une base de données XML

Dans cette section, nous allons montrer comment les documents XML ont été répartis et modifiés à des fins de stockage dans une base de données XML.

La répartition des documents est sensiblement identique à celle effectuée pour leur stockage dans un système de fichiers. La seule différence porte sur le lieu de stockage des schémas XML et des objets graphiques :

- Concernant les schémas XML, ils sont stockés automatiquement dans une zone de la base réservée aux schémas.
- Quant aux objets graphiques, nous choisissons de les stocker à l'extérieur de la base, dans l'un des sous-répertoires du serveur d'application que nous utilisons, à savoir `jakarta-tomcat-4.0.1`. On procède ainsi en raison de la grande simplicité de gestion de ces objets graphiques et de notre souci d'éviter des opérations de stockage/déstockage.

La figure 4-16 montre le résultat du stockage des fichiers de PiloteWeb dans une base de données XML.

La figure 4-35 présente le répertoire `Img` créé dans l'application PiloteWeb du serveur d'application jakarta-tomcat.

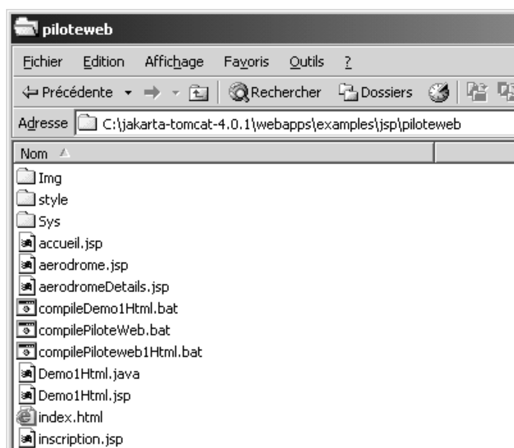


Figure 4–35
Lieu de stockage des illustrations de PiloteWeb dans le cas de la base de données XML

On considère ici que les illustrations sont des objets du même niveau que les feuilles de style de l'application. Elles sont stockées à l'extérieur de la base de données, mais toutefois dans une zone de l'application logique par rapport à son architecture. En clair, nous n'avons pas stocké ces données dans le premier répertoire utilisateur venu.

Là encore, les références vont devoir être changées :

- S'agissant des ID/IDREF utilisés entre les paquets XML dans le modèle logique originel, nous allons devoir les remplacer par des liens réellement pris en charge par les mécanismes de la base de données : ce sera XLink dans notre cas.
- S'agissant des illustrations, nous optons pour une reconstruction dynamique de leur URL d'adressage au moment de l'affichage. Nous n'allons donc garder à l'intérieur des données XML que le nom physique du fichier concerné. En effet, ayant assimilé ces objets à des éléments de l'application, il est logique que l'identification de leurs chemins d'accès soit prise en charge par l'application et non par les utilisateurs. Il serait illogique que ce chemin soit gravé en dur dans les données.

Nous montrons ci-après ce que deviennent les documents XML de PiloteWeb une fois adaptés aux considérations qui viennent d'être exposées.

Document XML pour les sites (dans l'attribut `nom`, nous ne précisons que le nom physique du fichier contenant l'objet graphique)

```
<?xml version="1.0" encoding="UTF-8"?>
<sites xmlns:xsi="…" xsi:noNamespaceSchemaLocation="sites.xsd">

 <site unique-id="p1">
  <nom>Nicolas le Panda</nom>
  <position x="48.7494" y="-2.1108"/>
  <photo nom="p1.gif" format="gif"/>
 </site>
</sites>
```

Document XML pour les pilotes (l'attribut `idrefSite` est remplacé par un élément link porteur des attributs XLink)

```
<?xml version="1.0" encoding="UTF-8"?>
<pilotes>
 <pilote unique-id="id11" publie="1">
  <link xlink:type="simple" xlink:show="embed"
        xlink:href="sites.xml/#xpointer(
                  ➥descendant::site[@unique-id='p1'])"/>
  <email>pilote@piloteWeb.com</email>
  <pwd>avion</pwd>
 </pilote>
</pilotes>
```

Document XML pour les aérodromes (l'attribut `idrefSite` est remplacé par un élément link porteur des attributs XLink)

```
<?xml version="1.0" encoding="UTF-8"?>
<aerodromes>
 <aerodrome unique-id="id13">
  <link xlink:type="simple" xlink:show="embed"
         xlink:href="sites.xml/#xpointer(
                   ➥descendant::site[@unique-id='a1'])"/>
  <frequence>125.90</frequence>
  <code>LFRM</code>
 </aerodrome>
</aerodromes>
```

Document XML pour les partenaires (l'attribut `idrefSite` est remplacé par un élément link porteur des attributs XLink)

```
<?xml version="1.0" encoding="UTF-8"?>
<partenaires>
 <partenaire unique-id="idpa15">
  <link xlink:type="simple" xlink:show="embed"
        xlink:href="sites.xml/#xpointer(
                   ➥descendant::site[@unique-id='pa1'])"/>
  <adresse>ZI des Arbihres Saint-Nazaire
  </adresse>
  <telephone>+33.2.35.60.42.89</telephone>
 </partenaire>
</partenaires>
```

Document XML pour les repères (l'attribut `idrefSite` est remplacé par un élément link porteur des attributs XLink)

```
<?xml version="1.0" encoding="UTF-8"?>
<reperes>
 <repere unique-id="id23">
  <link xlink:type="simple" xlink:show="embed"
        xlink:href="sites.xml/#xpointer(
                   ➥descendant::site[@unique-id='r1'])"/>
 </repere>
 <repere unique-id="id24">
  <link xlink:type="simple" xlink:show="embed"
       xlink:href="sites.xml/#xpointer(
                  ➥descendant::site[@unique-id='r2'])"/>
 </repere>
</reperes>
```

Document XML pour les aéroclubs (l'attribut `idrefSite` est remplacé par un élément link porteur des attributs XLink)

```
<?xml version="1.0" encoding="UTF-8"?>
<aeroclubs>
 <aeroclub unique-id="id17">
  <link xlink:type="simple" xlink:show="embed"
       xlink:href="partenaires.xml/#xpointer(
                  ➥descendant::partenaire[@unique-id='idpa15'])"/>
 </aeroclub>
 <aeroclub unique-id="id18">
  <link xlink:type="simple" xlink:show="embed"
        xlink:href="partenaires.xml/#xpointer(
                  ➥descendant::partenaire[@unique-id='idpa16'])"/>
 </aeroclub>
</aeroclubs>
```

Document XML pour les restaurants (l'attribut `idrefSite` est remplacé par un élément link porteur des attributs XLink)

```
<?xml version="1.0" encoding="UTF-8"?>
<restaurants>
 <restaurant unique-id="id19">
  <link xlink:type="simple" xlink:show="embed"
        xlink:href="partenaires.xml/#xpointer(
                  ➥descendant::partenaire[@unique-id='idre15'])"/>
  <horaires>Dès 7 heures et jusqu'à 20h. tous les jours sauf le WE
  </horaires>
 </restaurant>
</restaurants>
```

Document XML pour les loueurs (l'attribut `idrefSite` est remplacé par un élément link porteur des attributs XLink)

```
<?xml version="1.0" encoding="UTF-8"?>
<loueurs>
 <loueur unique-id="id21"
          href="partenaires.xml#idlo15">
  <link xlink:type="simple" xlink:show="embed"
        xlink:href="partenaires.xml/#xpointer(
                  ➥descendant::partenaire[@unique-id='idlo5'])"/>
 </loueur>
 <loueur unique-id="i32">
```

```
  <link xlink:type="simple" xlink:show="embed"
       xlink:href="partenaires.xml/#xpointer(
                 ➥descendant::partenaire[@unique-id='idlo6'])"/>
 </loueur>
</loueurs>
```

L'introduction de XLink nous oblige à modifier les schémas physiques individuels ainsi que le modèle logique des données. Tout dépend de la capacité de l'outil de modélisation à reconnaître la sémantique des liens de type XLink. S'il y parvient, le modèle physique pourra être produit automatiquement : dans le cas contraire, on devra chaque fois en passer par une retouche manuelle pour régénérer les schémas XML physiques.

En résumé…

Nous avons vu dans ce chapitre que XML laisse une grande liberté de choix dans la mise en œuvre physique des modèles de données. Si ses avantages sont la souplesse, la flexibilité et l'adaptabilité, il n'en est pas moins vrai que les modèles physiques sont *in fine* différents des modèles logiques.

La mise en œuvre physique de XML entraîne des modifications dans les modèles qui peuvent remonter jusqu'au modèle conceptuel. Il faut donc prévoir une nécessaire série d'itérations entre les modèles conceptuels, logiques et physiques dans le développement d'applications basées sur XML.

On retiendra que les facilités d'adressage dont est pourvu XML ont deux conséquences majeures sur les modèles de stockage physique :

- Ces derniers présentent des caractéristiques qui les différencient radicalement des modèles conceptuels : en partie grâce à la puissance de XQuery qui permet, le cas échéant, de reconstruire à la volée un DOM conforme au modèle conceptuel.
- Le modèle conceptuel peut être divisé en autant de sous-modèles indépendants que nécessaire. Cela n'empêche pas les données d'être liées entre elles.

5

Étape 5 - Finaliser la sémantique du balisage

Une fois le modèle de stockage réalisé, on peut dire que le modèle de données et sa transcription en XML sont maîtrisés. Un travail plus fin peut alors commencer. Il a pour objet d'optimiser le schéma XML.

Dans ce chapitre, nous allons présenter le travail d'optimisation qui consiste à étudier, élément par élément et attribut par attribut, la qualité de la sémantique transmise par ces objets aux applications.

Nous allons répondre aux questions suivantes :

- Quels noms définitifs donner aux éléments et attributs ?
- Comment faire passer plusieurs sémantiques par élément ?
- Quel ratio établir entre éléments et attributs ?
- Quelle est la sémantique d'un attribut par rapport à celle d'un élément ?
- Comment choisir entre attribut et sous-élément ?

Principes de base

Par le passé, on confondait souvent le fait de savoir lire le XML et celui de savoir lire les données. Il convient de revenir sur ce point pour le clarifier.

Pour nous aider dans cette tâche, nous allons considérer qu'un document XML présente toujours trois niveaux de lecture :

- Le **niveau syntaxique** : un programme comprenant la syntaxe XML sera toujours capable de lire un document XML, même en n'en comprenant pas la sémantique du balisage, et même sans avoir besoin d'en connaître le schéma. Qui plus est, ce programme sera capable d'opérer sur ce document lu « à l'aveugle » toute une série d'opérations de traitement, voire de transformation. Si les opérations de chargement en mémoire et de transformation sont parmi les plus importantes, il ne faut pas oublier pour autant la gestion des liens avec XLink, des inclusions avec Xinclude et de requêtage avec XQuery. Pour toutes ces opérations, seule une lecture de niveau syntaxique suffit : elle correspond au niveau InfoSet, le modèle de base des documents XML présenté au chapitre 2.
- Le **niveau structurel** : les programmes comprenant un schéma XML peuvent dire si un document est structurellement valide, c'est-à-dire conforme à la structure hiérarchique définie par le schéma qui lui est associé.
- Le **niveau sémantique** : les programmes comprenant la sémantique des objets XML savent leur appliquer des règles métier. Les données contenues dans le document XML peuvent dès lors être échangées intelligemment avec d'autres applications : on obtient un système d'information.

Les frontières qui existent entre ces niveaux doivent être connues et analysées, du moins à ce stade de la conception. Cependant, l'existence de ces niveaux laisse l'initiative aux développeurs d'applications : ils peuvent construire progressivement leur architecture de données.

Les éléments et attributs XML sont pris entre deux feux : d'un côté, ils doivent servir à contrôler la qualité des données qu'ils contiennent et, de l'autre, ils doivent être intelligemment reliés aux traitements. En même temps, un utilisateur doit pouvoir deviner quasi naturellement leur sens à la seule lecture. Quand un élément s'appelle `date`, on se doute bien qu'il s'agit d'une date, mais un élément de nom `datelc` (qui pourrait signifier *date limite de commande*) laisse rêveur.

Une partie de la flexibilité et de l'évolutivité demandées aux systèmes d'information modernes se joue sur la lisibilité du XML qu'ils traitent, notamment en ce qui concerne le niveau sémantique : il est ainsi possible de développer des programmes ne traitant qu'une partie des données contenues dans un document XML, ou de concevoir des noms d'éléments satisfaisant aux exigences de nommage de plusieurs applications.

La question du nommage des éléments et attributs XML est comme un curseur qu'il faudrait positionner entre deux extrêmes représentant deux qualités différentes : d'un côté, une bonne représentation de la donnée et, de l'autre, une bonne représentation des traitements, comme cela est illustré à la figure 5-1.

Figure 5–1
Le nom d'un objet XML peut représenter deux types opposés de sémantique.

Par exemple, une structure date constituée de la seule balise `date` peut suffire, comme dans `<date>23 mai 1959</date>`, mais si l'on veut traiter le contenu de cette date, il vaudra mieux utiliser trois sous-éléments pour les jour, mois et année, tels que :

```
<date><jour>23</jour><mois>mai</mois><année>1959</année></date>
```

ou encore

```
<date><jj>23</jj><mm>05</mm><aa>1959</aa></date>
```

Dans cette toute dernière forme, on a remplacé le nom du mois par son numéro : si cette date est de ce fait moins facile à lire naturellement, elle y gagne cependant en capacité de traitement.

C'est toutes ces questions que nous étudions dans les sections suivantes de ce chapitre.

Type et sémantique

Type et sémantique désignent deux caractéristiques complémentaires d'un objet.

Le *type* d'une donnée est sa nature physique : une date, un nombre décimal, entier, réel, positif ou négatif, une chaîne de caractères, etc. sont autant de types qui fournissent plus ou moins directement des indications sur la nature de l'information. Ces types permettent de la reconnaître et de la contrôler. En XML, un type est simple (c'est-à-dire que la donnée s'écrit directement sans avoir besoin de sous-éléments, un nombre entier par exemple) ou complexe (des sous-éléments sont nécessaires pour décrire la donnée, c'est le cas de notre balise date ci-dessus).

La *sémantique* est une information complémentaire qui définit la signification applicative de la donnée. De quelle date s'agit-il ? Qu'est-ce qu'elle détermine ? À quoi sert-elle ? Écrire `<date>23 mai 1959</date>` ou `<date>vingt-trois mai mille`

neuf cent cinquante neuf</date> permet à la seule lecture de savoir qu'il s'agit d'une date, mais préciser <date-de-naissance>23 mai 1959</date-de-naissance> indique à coup sûr qu'il s'agit d'une date de naissance, même en ignorant tout du schéma qui valide un tel document XML.

On le vérifie, XML a toujours offert une bonne expression de la sémantique, mais pas du type. En comparaison, le couple (colonne, type) du relationnel fait correspondre une sémantique (le nom de la colonne) à son type, par exemple colonne Date_de_naissance et type date. Cependant, cette correspondance (sens, type) est limitée par l'organisation en deux dimensions du modèle relationnel (table, colonne). Les emboîtements des éléments XML offrent infiniment plus de souplesse, en particulier depuis l'introduction des schémas.

En effet, grâce aux travaux réalisés pour XML Schema, le type des éléments peut désormais non seulement être complémentaire de la sémantique mais lui être associé implicitement ou explicitement. On utilise donc les noms d'éléments de son choix, tels que date-de-naissance, mais on spécifie de plus que cette date de naissance est d'un certain type. On peut ainsi la faire figurer selon une forme prédéfinie : celle du type (par exemple sous la forme 19590523). Il n'est dès lors plus besoin de sous-éléments pour que les programmes sachent à la fois contrôler la validité de cette donnée et la manipuler en tant que date.

Exploiter le type et la sémantique

Exprimer le type et la sémantique grâce aux schémas XML est une chose, les exploiter en est une autre.

Comme nous l'avons dit, XML transporte naturellement la sémantique : la seule lecture des éléments et des attributs suffit en général. Cependant, XML ne transporte pas naturellement le type.

On est confronté aux quatre cas de figure suivants.

Cas n°1 : le document XML est simplement bien formé. Il n'y a aucun transport des types. Seule la sémantique des données apparaît au travers des noms des éléments.

```
<?xml version="1.0"?>
<partenaires>
 <partenaire>
  <nom>La bonne étoile</nom>
  <horaires>
   <ouverture>7</ouverture>
```

```
    <fermeture>20</fermeture>
    <jours>12345</jours>
   </horaires>
   <adresse>
    <rue>ZI des Arbihres</rue>
    <ville>Saint-Nazaire</ville>
   </adresse>
   <telephone>
    <pays>33</pays>
    <region>2</region>
    <numero>35604289</numero>
   </telephone>
  </partenaire>
</partenaires>
```

Cas n°2 : on transmet le type des données via un attribut intitulé `type`. Une application réceptrice sera donc en mesure de contrôler le type des données. Le type ainsi transmis s'exprime en utilisant les noms standards d'un catalogue connu (ici, nous avons utilisé le catalogue des types de base de XML Schema) ou ceux d'un catalogue spécifique à l'application.

```
<?xml version="1.0"?>
<partenaires xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <partenaire>
  <nom xsi:type="xs:string">La bonne étoile</nom>
  <horaires>
   <ouverture xsi:type="xs:int">7</ouverture>
   <fermeture xsi:type="xs:int">20</fermeture>
   <jours xsi:type="xs:int">12345</jours>
  </horaires>
  <adresse>
   <rue xsi:type="xs:string">ZI des Arbihres
   </rue>
   <ville xsi:type="xs:string">Saint-Nazaire
   </ville>
  </adresse>
  <telephone>
   <pays xsi:type="xs:int">33</pays>
   <region xsi:type="xs:int">2</region>
   <numero xsi:type="xs:int">35604289</numero>
  </telephone>
 </partenaire>
</partenaires>
```

Cas n°3 : le document XML fait référence au schéma XML qui en contrôle la validité. La connaissance du type des données nécessite de savoir lire un schéma XML et que ce dernier soit accessible..

```
<?xml version="1.0"?>
<partenaires xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="partenaires.xsd">
 <partenaire>
  <nom>La bonne étoile</nom>
  <horaires>
   <ouverture>7</ouverture>
   <fermeture>20</fermeture>
   <jours>12345</jours>
  </horaires>
  <adresse>
   <rue>ZI des Arbihres</rue>
   <ville>Saint-Nazaire</ville>
  </adresse>
  <telephone>
   <pays>33</pays>
   <region>2</region>
   <numero>35604289</numero>
  </telephone>
 </partenaire>
</partenaires>
```

Cas n°4 : voici un extrait du PSVI de notre document `partenaires.xml`. Le PSVI est un document XML qui contient l'intégralité du document XML originel et son schéma associé. Ici, nous voyons que pour le seul élément `ouverture`, on obtient un nombre important d'informations parmi lesquelles : son contenu initial, son type et la forme lexicale normalisée de son contenu. Les éléments et données du document XML originel sont complètement encapsulés par un nouveau type de balisage, en cours de normalisation, qui est celui du PSVI. Cette forme a l'avantage de transporter toute l'information au sein d'un même et seul document XML… Il n'est toutefois pas évident qu'elle soit simple à lire par les utilisateurs. On a ici un maximum d'information de typage en conservant la sémantique du balisage initial et l'exploitation d'un tel document XML par des programmes est optimale..

```
<element baseURI="file:///C:/exemplesXSV/partenaires.xml"
localName="ouverture"
nil="false"
p:schemaNormalizedValue="7"
p:validationAttempted="full" p:validationContext="g1"
p:validity="valid">
```

```
<text content="7"/>
<namespace namespaceName=
➥"http://www.w3.org/XML/1998/namespace"
prefix="xml"/>
<namespace namespaceName=
➥"http://www.w3.org/2001/XMLSchema-instance"
prefix="xsi"/>
<p:atomic ref="xsd..type.int" i:nil="true"/>
</element>
```

CONCEPT PSVI

On note PSVI pour Post Schema Validation InfoSet. Il s'agit d'un ensemble d'informations obtenu par la validation de schéma. Il contient la totalité du document XML parsé et la totalité des informations issues du schéma ou calculées pendant l'opération de validation (exemple : les valeurs lexicales canoniques). Bien qu'informelle, vous trouverez une définition précise du PSVI à l'URL http://www.w3.org/2001/05/serialized-infoset-schema.html et du schéma XML correspondant à l'adresse http://www.w3.org/2001/05/PSVInfoset.xsd.

VOCABULAIRE Validation de schéma

L'expression validation de schéma désigne la validation d'un document XML eu égard à son schéma XML.

VOCABULAIRE Forme lexicale normalisée, ou canonique

La forme lexicale normalisée d'une donnée désigne sa représentation lexicale (c'est-à-dire sa forme textuelle dans un fichier) obtenue quand on tient compte du type de la donnée. Le cas le plus typique est la représentation des nombres : si on écrit `+003.1200` dans un document XML et que l'on a déclaré cette donnée comme étant de type décimal, sa forme lexicale normalisée sera 3.12. Le tome 2 de la recommandation XML Schema précise les formes normales lexicales des types acceptés par XML Schema. Vous trouverez cette recommandation à l'URL http://www.w3.org/TR/xmlschema-2/.

Manières d'exprimer la sémantique

Pour ce sujet, nous allons partir d'exemples que nous décrivons brièvement.

Exemple 1

```
<h1>La bonne étoile</h1>
<p>Horaires : Ouvert de 7 heures à 20h, tous les jours sauf le WE</p>
<p>Adresse : ZI des Arbihres Saint-Nazaire</p>
<p>Tél : +33.2.35.60.42.89</p>
```

Les éléments `p` et `h1` sont traditionnellement utilisés pour marquer des paragraphes et des titres de niveau 1 dans des documents (et en particulier en HTML).

Il s'agit typiquement de balises qui n'informent que très peu, voire pas du tout sur la nature du contenu textuel qu'elles transportent. Il s'agit d'un balisage dit « de type document ».

Leur rôle essentiel est de provoquer des ordres de composition particulier : une mise en exergue pour les titres, leur renvoi en table des matières et une composition simple pour les paragraphes…

La sémantique de ces éléments est mono-applicative, à savoir que cela ne concerne que les seules applications de composition des documents, de mise en page.

Exemple 2

```
<h1 xsi:type="nom-du-partenaire">La bonne étoile</h1>
<p xsi:type="horaires-ouverture">Ouvert de 7 heures à 20h, tous les
jours sauf le WE</p>
<p xsi:type="adresse">ZI des Arbihres Saint-Nazaire</p>
<p xsi:type="telephone">+33.2.35.60.42.89</p>
```

En ajoutant l'attribut `type`, on apporte une information relative au type de contenu.

Le document XML devient nettement plus compréhensible pour le lecteur humain et la sémantique des données est transportée jusqu'à l'application de composition. Cette dernière pourra, à sa guise, l'utiliser ou l'ignorer.

On peut supprimer une partie du contenu textuel et laisser le soin à l'application de produire des textes complémentaires tels que les préfixes *Horaires :*, *Adresse :* ou encore *Tél :*. En rendant le contenu textuel linguistiquement plus neutre, on facilite l'échange entre systèmes et les opérations de transformation.

Exemple 3

```
<h1 xsi:type="nom-du-partenaire">La bonne étoile</h1>
<p xsi:type="horaires-ouverture"><var varType="ouverture">7</var>
<var varType="fermeture">20</var><var varType="jours">12345</var> </p>
<p xsi:type="adresse"><var varType="rue">ZI des Arbihres</var>

<var varType="ville">Saint-Nazaire</var></p>
<p xsi:type="telephone"><var varType="pays">33
</var><var varType="region">2</var><var varType="numero">35604289
</var></p>
```

Ici on conserve le même balisage principal (les `h1` et `p` de l'application de composition) et on développe un sous-balisage pour décrire la sémantique des données.

L'élément `var` sert de fourre-tout transportant la sémantique des données jusqu'à l'application de composition, à qui revient la charge de construire des phrases compréhensibles pour le lecteur.

L'intérêt ici, c'est de pouvoir développer des applications multilingues à partir d'un même fonds de données ; en revanche, il n'est pas aisé de savoir quelle partie de l'application doit opérer la transformation du contenu. Cette question sera abordée au chapitre 7.

Exemple 4

```
<partenaire>
 <nom>La bonne étoile</nom>
 <horaires>
  <ouverture>7</ouverture>
  <fermeture>20</fermeture>
  <jours>12345</jours>
 </horaires>
 <adresse>
  <rue>ZI des Arbihres</rue>
  <ville>Saint-Nazaire</ville>
 </adresse>
 <telephone>
  <pays>33</pays>
  <region>2</region>
  <numero>35604289</numero>
 </telephone>
</partenaire>
```

Ici, le balisage est exclusivement orienté données. Tout le balisage de type présentation (les éléments `p` et `h1`) a disparu.

L'application doit prendre en charge une transformation complète des données *et* des éléments XML : le texte final doit être synthétisé à partir de la sémantique induite par les noms des éléments XML qui doivent être mis en correspondance (transformés) avec des éléments de présentation.

La transformation des éléments de ce document XML en éléments de présentation (en HTML par exemple) peut ne pas être évidente.

La sémantique des données ne passe que par les seuls éléments XML que l'application doit comprendre dans leur intégralité et individuellement.

Exemple 5

```
<partenaire>
 <nom>La bonne étoile</nom>
<heure-ouverture>7</heure-ouverture>
<heure-fermeture>20</heure-fermeture>
 <adresse>…</adresse>
 <telephone>…</telephone>
</partenaire>
```

ou

```
<partenaire>
 <nom>La bonne étoile</nom>
<heure-ouverture-en-semaine>7
</heure-ouverture-en-semaine>
<heure-fermeture-en-semaine>20
</heure-fermeture-en-semaine>
 <adresse>…</adresse>
 <telephone>…</telephone>
</partenaire>
```

ou

```
<partenaire>
 <nom>La bonne étoile</nom>
<heure-ouverture-en-semaine-en-vacances>7
</heure-ouverture-en-semaine-en-vacances >
<heure-fermeture-en-semaine-en-vacances >20
</heure-fermeture-en-semaine-en-vacances >
 <adresse>…</adresse>
 <telephone>…</telephone>
</partenaire>
```

Dans ce dernier exemple, nous montrons qu'à un élément peut être attachée une sémantique plus ou moins précise *via* son nom ou un attribut. Les noms d'éléments `heure-ouverture`, `heure-ouverture-en-semaine` et `heure-ouverture-en-semaine-en-vacances` sont, dans cet ordre, classés du plus vague au plus précis.

Au travers de ces exemples, on montre la diversité des cas de figure, depuis un balisage orienté *document* ressemblant à du HTML jusqu'à des balisages orientés *données* mettant en valeur soit la sémantique, soit les types, ou bien les deux.

Ces nombreuses possibilités de XML indiquent qu'il faut essayer dans la mesure du possible de choisir les noms et la structure des éléments en tenant compte de leur contexte d'utilisation : ainsi réservons-nous au monde du document des modèles de balisage orientés document et au monde du calcul ceux orientés données.

Adapter les structures au contexte

On comprend que la structure XML des exemples fournis à la section précédente (à l'exception du premier cas) transporte peu ou prou la même sémantique. Finalement, il semble bien que seule change la forme.

La différence entre toutes ces formes de balisage tient dans leur adéquation avec leur contexte d'utilisation : quels sont les balisages et vocabulaire bien adaptés ?

Le contexte d'utilisation est la partie de l'application qui va être concernée par les données suivantes : est-ce l'affichage qui est en question ? la composition ? le transport ? la transformation ? le stockage ?

Comme cela apparaît à la figure 5-2, il est difficile de parler d'un seul et unique traitement. Tout au long de leur cycle de vie, les données XML vont être au service de plusieurs applications ainsi que nous l'avons évoqué dès le chapitre 1.

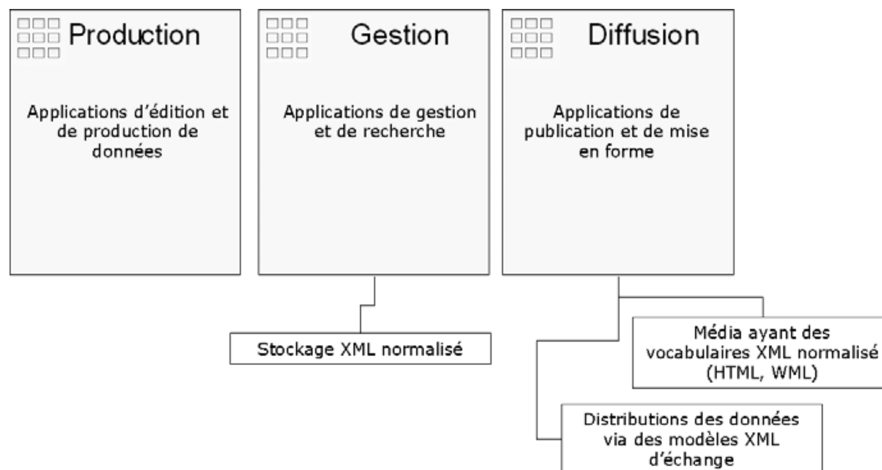


Figure 5–2 La forme XML doit être adaptée au contexte d'utilisation.

À l'examen de cette figure, on comprend mieux pourquoi il est impossible d'imaginer qu'une seule forme de balisage XML suffira à satisfaire tous les cas de figure car la

mise en œuvre de XML s'accompagne obligatoirement de développements de programmes de transformation de l'information.

Parmi les formes montrées à la section précédente, certaines conviennent mieux au stockage (les cas 4 et 5 qui exposent bien la sémantique), d'autres à l'affichage (cas 1 qui utilise un balisage orienté document) et enfin d'autres encore au transport (cas 2 et 3 qui exposent prioritairement le typage des données). S'il existe des tendances générales en ce qui concerne ces formes, on ne peut toutefois affirmer *in abstracto* que telle forme est meilleure que telle autre : il faut toujours tenir compte du contexte applicatif.

On retiendra toutefois les directives suivantes :

- Au niveau du stockage, tant le nom des éléments que leur hiérarchie ne doivent pas dépendre directement d'un programme en particulier. Par exemple, chercher à établir des équivalences directes entre noms d'éléments XML et ceux de classes Java est un (très) mauvais choix conceptuel – nous en expliquerons les raisons dans la section suivante.
- Le modèle de stockage doit être le plus représentatif possible du modèle conceptuel des données, lui-même potentiellement différent de celui nécessaire aux applications. L'affichage en HTML d'un tableau de données en est un exemple caractéristique : il est évidemment très réducteur de stocker dans une base de données XML ledit tableau sous sa forme HTML. Il est préférable de procéder à l'aide d'un balisage « informatif » qui fera ressortir la logique des données à l'intérieur du tableau. En d'autres termes, pour décrire un tableau de boulons, il est plus puissant d'utiliser les balises <boulon>, <diamètre>, <longueur>, <code_article>… que les <table>, <tr>, <td> du langage HTML.
- Afin de ne pas trop rigidifier le modèle de stockage au travers des structures d'éléments, vous pouvez faire passer tout ou partie de la sémantique par des valeurs d'attributs.

Outrepasser le principe des objets métier Java

Le principe des objets métier réside dans la mise en correspondance des noms d'éléments XML et des classes Java qui en assurent le traitement. De nombreuses raisons nous amènent à critiquer cette manière de concevoir les applications :

- Il est dangereux de lier deux mondes aussi différents que ceux des données et des traitements. Rendus dépendants, ni le modèle de données, ni les traitements ne peuvent plus évoluer librement, ce qui serait contraire à l'esprit de XML et de toute architecture applicative évolutive.
- En XML, la signification des noms d'éléments peut être contextuelle. Dans nos exemples, l'élément `horaire` a différentes sémantiques selon que ses parents sont `ouverture`, `fermeture` ou encore `vacances`. On peut supposer que les traitements

appliqués aux données seront différents dans chacun de ces trois cas. Il faut pouvoir associer plus d'un programme à l'élément `horaire`.

- En XML, un seul nom d'élément ne suffit pas à un programme : la connaissance des valeurs d'attributs, sous-éléments, voire éléments parents, peut se révéler décisive pour les programmes de traitement. Comment faire lorsque seul le nom de l'élément est associé à un, et un seul, programme ?
- Associer brutalement un nom d'élément à une classe bloque toute évolution du schéma XML. En effet, lors de la révision d'un schéma, un nom d'élément peut être remplacé par un autre, changé en attribut, déplacé dans la structure, et il peut même être transformé sémantiquement. Il faut que ces modifications soient faites tout en conservant les anciens traitements : anciens et nouveaux programmes doivent cohabiter malgré les modifications apportées au balisage.
- L'inverse est vrai, il faut parfois disposer de plusieurs programmes différents pour un même élément, le choix entre les traitements pouvant par exemple intervenir au vu de la valeur d'un attribut.

La bonne architecture consiste donc à utiliser des techniques associant de manière logique les éléments aux programmes de traitement. Y procéder par le seul jeu des noms d'éléments est voué à l'échec. Les éléments peuvent être associés aux programmes de traitement par le biais d'un tableau comportant quatre colonnes : une première colonne sert à lister un ou plusieurs nom(s) d'élément(s), une deuxième colonne sert à spécifier des conditions contextuelles pour ces éléments, dans les troisième et quatrième colonnes on spécifie le programme à exécuter quand la balise ouvrante (respectivement fermante) est rencontrée.

HISTOIRE CALS

CALS est l'acronyme de Computer Aided Logistics Support. Ce projet du département de la Défense américaine est historiquement celui qui permit, en 1988, le décollage de SGML. Son impact sur l'essor de SGML fut tel que le mot CALS est devenu à une époque synonyme de SGML. Un peu comme Frigidaire était synonyme de réfrigérateur. Il en est resté un modèle de tableau dit « le modèle CALS » que tous les éditeurs SGML/XML mettent en œuvre nativement.

Dans l'exemple du tableau 5-1, nous présentons un exemple typique de spécification de traitement d'un flot XML. Les éléments XML concernés sont ceux qui définissent un tableau selon le modèle CALS. Cet exemple présente tous les cas de figure que nous avons énumérés plus haut, à savoir :

- L'élément n'est associé à aucun traitement. C'est le cas des éléments `table`, `thead`, `tbody` et `tfoot`.
- Les éléments subissent tous le même traitement. C'est le cas des éléments `table`, `thead`, `tbody` et `tfoot` qui ont tous été écrits dans une même cellule du tableau.

- Les éléments subissent un traitement conditionné à un contexte. C'est le cas des éléments `title`, `row` et `entry`, dont les différents contextes apparaissent dans la deuxième colonne sous la forme d'expressions Xpath.
- Les éléments associés à des traitements tant sur la balise ouvrante que sur la balise fermante. C'est le cas des éléments `tgroup`, `row` et `entry`.
- Les valeurs attributs de l'élément ou de ses parents modifient le traitement à appliquer. C'est le cas de l'élément `entry`.

Tableau 5–1 Exemple de spécification de traitement d'un flot XML

ÉLÉMENTS SELON LE CONTEXTE			
Élément	**Contexte**	**Balise ouvrante**	**Balise fermante**
<table> <thead> <tbody> <tfoot>			
<title>	Table/title	Utiliser le style `title:table` Ce style doit chevaucher les colonnes si l'attribut `pgwide` est présent sur l'un des éléments parent de `title`.	
<tgroup>		Inverser les branches enfant `tfoot` et `tbody`. Insérer un tableau dans le document ayant le nombre de colonnes spécifié par l'attribut `cols`, chevauchant les colonnes de la page si l'attribut `pgwide` a été utilisé… Et on peut trouver ici toute une série de spécifications de mise en forme du tableau dépendant des valeurs des attributs trouvés sur cette balise dans le document XML.	Arrêter le tableau.
<colspec>		Reconnaître les caractéristiques des colonnes. Les stocker pour les utiliser ensuite dans la fabrication du tableau final.	
<spanspec >		Stocker les caractéristiques de fusion de cellules définies via cet élément.	
<row>	thead/row	Insérer une rangée d'en-tête qui se répète sur chaque page.	Terminer la rangée.
<row>	tfoot/row	Insérer une rangée de pied de tableau.	Terminer la rangée.
<row>	tbody/row	Insérer une rangée normale.	Terminer la rangée.
<entry>	./[@align= "char"] ./@namest	Insérer une cellule contenant un paragraphe p dans la colonne spécifiée par la valeur de l'attribut `namest`. Aligner le texte sur le caractère spécifié par l'attribut `charoff`.	Terminer la cellule.

Tableau 5–1 Exemple de spécification de traitement d’un flot XML

ÉLÉMENTS SELON LE CONTEXTE			
Élément	**Contexte**	**Balise ouvrante**	**Balise fermante**
`<entry>`	./@namest	Insérer une cellule contenant un paragraphe p dans la colonne spécifiée par la valeur de l’attribut `namest`.	Terminer la cellule.
`<entry>`	./[@align= "char"] ./@spannames	Insérer une cellule contenant un paragraphe p. Aligner le texte sur le caractère spécifié par l’attribut `charoff`. Fusionner la cellule selon la spécification `spannames`.	Terminer la cellule.
`<entry>`	./@spannames	Insérer une cellule contenant un paragraphe p. Fusionner la cellule selon la spécification `spannames`.	Terminer la cellule.
`<entry>`	./[@align= "char"]	Insérer une cellule contenant un paragraphe p. Aligner le texte sur le caractère spécifié par l’attribut `charoff`.	Terminer la cellule.
`<entry>`		Insérer une cellule contenant un paragraphe p.	Terminer la cellule.

Cet exemple montre bien comment, en XML, il est nécessaire de pouvoir établir une carte d’association, parfois sophistiquée, entre éléments et traitements associés.

Nous avons pris ici le cas du balisage d’un tableau ; c’est certes un cas typique, mais cela eût été la même chose avec tout autre type de données.

Utilisation des attributs pour passer la sémantique

En général, l’introduction de la sémantique par les noms d’éléments réduit considérablement les possibilités d’évolution des schémas. Nous allons en expliquer les raisons.

L’intérêt qu’il y a à transmettre la sémantique des données est évidemment de faire passer à un programme récepteur une information la plus précise possible, et donc d’avoir un grand choix. Si on décide de faire passer la sémantique par les éléments, tout nouveau nom devra être formellement déclaré dans la DTD ou le schéma XML associé au document : croyant enrichir la sémantique du document XML, on touche en réalité sa structure. Or, cela n’est pas la même chose : structure et sémantique sont deux dimensions différentes.

En utilisant les attributs, on fait varier la sémantique des données sans toucher à leur structure. On peut dès lors exprimer un grand nombre de possibilités sans avoir à écrire autant de schémas. On diminue la complexité des applications, le nombre de révisions des DTD et les risques liés à la cohabitation des schémas.

Si l'on peut disposer de plusieurs sémantiques à schémas égaux, alors l'introduction d'une nouvelle sémantique ne touchera que très localement les programmes déjà écrits, ce qui n'est pas le cas quand on change une structure.

Quand on fait passer la sémantique par le nom de l'élément, une seule sémantique est exprimée. Quand on utilise les attributs, on fait passer autant de sémantiques que l'on veut. Nous donnons des exemples ci-après.

Exemple 1 : Cas où il n'y a pas d'attribut.

```
<heure-ouverture>8h30 tous les jours sauf le samedi où l'on ouvre à 7h30
</heure-ouverture>
```

ou

```
<heure-ouverture>8.30</heure-ouverture>
```

La sémantique passe uniquement par le nom de l'élément. Tout changement de sémantique entraînera une révision de la DTD ou du schéma XML de l'application.

Remarquons toutefois qu'entre les deux cas de figure présentés ici, et utilisés tout au long de ce tableau, la précision des données transmises n'est pas identique.

Exemple 2

```
<heure-ouverture contenuType="texte"> 8h30 tous les jours sauf le samedi
où l'on ouvre à 7h30</heure-ouverture>
```

ou

```
<heure-ouverture contenuType="numérique">8.30</heure-ouverture>
```

Admettons que l'on souhaite autoriser la saisie d'un texte libre ou d'une heure à l'intérieur de l'élément. Admettons que cette information soit susceptible de passer au travers d'un programme de traitement qui aura un comportement différent selon qu'il détecte une expression textuelle ou une indication numérique d'heure.

Nous avons alors tout intérêt à ajouter un attribut, comme nous l'avons fait ici avec l'attribut `contenuType`, permettant de spécifier le type « applicatif » de cette information. En ne limitant pas notre modèle aux seuls types de XML Schema, on signifie que l'on se réserve le droit de créer son propre catalogue de types. La charge incombe ensuite aux programmes d'en contrôler la cohérence. Cela n'est pas un mal, le catalogue des types de XML Schema est souvent trop restrictif pour nous en contenter.

Exemple 3

```
<heure roleType="ouverture" contenuType="texte">8h30 tous les jours
sauf le samedi où l'on ouvre à 7h30
</heure>
```

ou

```
<heure roleType="ouverture" contenuType="numérique">8.30</heure>
```

Ici, nous faisons la distinction entre un attribut qui va qualifier l'élément XML et un attribut qui va qualifier l'usage de la donnée pour une application.

L'attribut `roleType` vient qualifier l'élément `heure-ouverture`, désormais réduit à un nom très générique. Ici, on précise qu'il s'agit d'une heure d'ouverture.

L'attribut `contenuType` vient qualifier la donnée pour l'application qui la traite. Ici, on l'utilise pour préciser s'il s'agit d'une chaîne de caractères ou d'une valeur numérique.

Exemple 3

```
<heure roleType="ouverture" contenuType="numérique"
applicType="LMMJVD">8.30</heure>
<heure roleType="ouverture" contenuType="numérique"
applicType="S">7.30</heure>
```

Afin de faire passer exactement le même niveau d'information entre les deux cas de figure qui servent de support à nos exemples, nous ajoutons désormais un attribut, `applicType`, permettant de préciser le domaine d'applicabilité de l'élément `heure`.

Ici, on précise dans le premier cas que l'heure d'ouverture est applicable du lundi au dimanche sauf le samedi, et dans le second que l'heure d'ouverture précisée ne s'applique que le samedi.

TECHNIQUE Notion d'applicabilité

La notion d'applicabilité des données est probablement aussi importante que celle de type. En effet, la justesse d'une donnée dépend parfois de son contexte d'utilisation. Dans le domaine mécanique, il existe de nombreux cas de données vraies dans un certain contexte et fausses dans d'autres. Le secteur automobile est riche de ces cas puisque les constituants d'une voiture varient énormément d'un modèle à un autre, en fonction de l'endroit, par exemple, où la voiture est fabriquée. L'applicabilité consiste à indiquer le contexte de validité d'une information. Parler de domaine de validité d'une donnée reviendrait au même.

Le balisage proposé dans le dernier des exemples précédents offre à la fois souplesse, précision et liberté d'adaptation :

- souplesse en raison du très grand nombre de cas de figure qu'il prend finalement en compte ;
- précision car plusieurs sémantiques sont transmises ;
- liberté d'adaptation car de nouvelles valeurs et possibilités sont très facilement utilisées.

Enfin, ce type de balisage est lui-même générique, et l'expérience montre qu'il peut être généralisé à presque tous les éléments d'une DTD ou d'un schéma.

Ainsi, quand on conçoit un schéma XML, on peut toujours partir du principe que les éléments se verront attribuer systématiquement trois propriétés :

- un attribut de précision pour affiner la sémantique portée par le nom de l'élément, ici, l'attribut `roleType` ;
- un attribut de spécification du domaine de validité de l'élément, ici l'attribut `applicType` ;
- un attribut de spécification destiné à l'application qui reçoit les données, ici l'attribut `contenuType`.

En ce qui concerne les valeurs autorisées pour ces attributs, il n'y a pas de règle lexicale particulière. C'est à vous de définir les règles qui conviennent à vos propres cas de figure.

En résumé…

Dans ce chapitre, nous avons abordé le problème de la relation des éléments XML avec les programmes qui doivent les traiter.

Nous avons vu que la mise en correspondance directe des noms d'éléments XML avec ceux des programmes qui les traitent est une mauvaise conception.

Nous avons expliqué la différence entre structure et sémantique, et proposé plusieurs solutions pour transmettre la sémantique des éléments aux applications.

En faisant attention à bien dissocier les données des traitements et en utilisant des formes XML bien choisies, il est possible d'augmenter sensiblement la souplesse et la pérennité des applications.

Au chapitre suivant, nous allons étudier le cas d'utilisation des variantes de modèles. D'une part, nous allons montrer pourquoi ces variantes sont presque toujours nécessaires, et d'autre part en quoi elles ne doivent pas créer d'interférences dans les applications.

6

Étape 6 - Produire les variantes des schémas XML

Nous allons maintenant aborder la façon dont il convient de gérer les variantes des modèles de base. À ce stade de la conception d'un modèle, les schémas XML sont suffisamment connus et stabilisés pour que les conséquences de leurs évolutions futures puissent être envisagées.

Il serait en effet réducteur d'imaginer que les modèles XML n'évolueront pas une fois l'application développée. La souplesse de XML, qui permet de faire cohabiter plusieurs versions d'un même modèle au sein d'une même application, est en général exploitée.

Pourquoi des variantes ?

Deux séries de raisons justifient l'existence de variantes. Une première est fournie par XML lui-même tandis que la seconde provient de la nature de la matière manipulée : l'information.

Raisons liées à la nature de XML

Un document XML est loin d'avoir une seule forme et peut exister sous des formes aussi différentes que :

- Un document de base bien formé résultant d'une saisie manuelle ou d'une génération automatique.

- La forme canonique XML du document XML de base : celle dans laquelle le parseur a par exemple ajouté les valeurs d'attributs par défaut ou imposées.
- La forme canonique lexicale du document XML de base : celle dans laquelle le parseur a modifié le contenu des éléments et attributs en tenant compte des types simples qui leur ont été affectés *via* le schéma XML. Les blancs sont par exemple réduits quand ils sont en surnombre dans les chaînes de caractères, les zéros de tête et de queue sont écrêtés dans les nombres, les booléens *0* et *1* remplacés par les mots *false* et *true*, etc.
- La forme consolidée, ou expansée, du document XML de base : les éventuels sous-documents inclus par XInclude sont alors physiquement inclus, les entités textuelles sont remplacées par leur contenu, etc.
- La forme PSVI, pour Post Schema Validation InfoSet, du document XML de base. Il s'agit d'une version du document XML dans laquelle on introduit la totalité des définitions déductibles du schéma associé.

Raisons liées à la nature de l'information

La seconde série de raisons est relative aux principes mêmes que devrait respecter tout système d'information : souplesse, adaptabilité, capacité à être connecté à d'autres systèmes, etc. Autrement dit, elle est relative à :

- l'évolution naturelle des fonctions du système ;
- l'extension du jeu de données pris en charge par l'application ;
- l'amélioration de la qualité des données ou des formats d'échange de données entre applications ;
- la prise en charge de différentes demandes spécifiques de clients (au sens commercial du terme « client ») ;
- les règles métier spécifiques d'un projet ;
- le besoin de disposer de plusieurs formats de stockage (pour différents médias) ;
- le besoin de permettre à l'application de manipuler et stocker momentanément des formats XML légèrement différents des formats officiels. Par exemple, pouvoir stocker des documents « bien formés », mais pas encore valides…
- l'enrichissement des structures XML par des métadonnées de gestion ou de traçabilité de la qualité ;
- la correction de bogues du modèle : en XML, il arrive que les modèles soient bogués quand, par exemple, on s'aperçoit à l'usage que telle structure est impossible à réaliser alors qu'elle devrait l'être, tandis que d'autres sont interdites quand elles devraient être autorisées.

Un cas concret

Nous allons utiliser dans ce chapitre le cas concret de PiloteWeb. Nous décidons de modifier le schéma `sites.xsd` afin que les utilisateurs puissent ajouter un prénom à l'intérieur du champ `nom`.

Nous souhaitons indifféremment pouvoir obtenir des documents XML de la forme :

```
<si:sites>
<si:site unique-id="ID000000">
  <si:nom>Nicolas le Panda</si:nom>
  <si:photo nom="p1" format="gif"/>
  <si:position>
   <si:x>48.7494</si:x>
   <si:y>-2.1108</si:y>
  </si:position>
</si:site>
</si:sites>
```

ou de la forme :

```
<si:sites>
<si:site unique-id="ID000000">
  <si:nom><si:prénom>Nicolas</si:prénom> le Panda</si:nom>
  <si:photo nom="p1" format="gif"/>
  <si:position>
   <si:x>48.7494</si:x>
   <si:y>-2.1108</si:y>
  </si:position>
</si:site>
</si:sites>
```

Nous allons étudier les différentes manières d'identifier les schémas correspondant à ces variantes.

Comment identifier les variantes ?

Qui dit variantes, dit schémas différents ! La manière de gérer l'identification de ces schémas dépendra des deux cas de figure suivants :

- soit le schéma définit un vocabulaire sans espace de noms cible ;
- soit le schéma définit un vocabulaire dans un espace de noms cible.

Nous allons successivement étudier l'un et l'autre cas. Dans le texte ci-après, nous utilisons les expressions de schéma initial et de schéma variant.

Schéma sans espace de noms cible

Si le vocabulaire du schéma initial n'a pas été projeté dans un espace de noms, celui du schéma variant ne le sera pas non plus. Il suffit dès lors d'adapter l'attribut `noNamespaceSchemaLocation` porté par les éléments racines des documents XML concernés.

Dans l'exemple suivant, le premier document XML référence le fichier contenant `sites.xsd`, tandis que le second référence la variante de ce schéma contenue dans le fichier `sitesVariante.xsd`.

Voici le document XML référençant le schéma initial :

```
<?xml version="1.0" encoding="UTF-8"?>
<sites xmlns:xsi="…" xsi:noNamespaceSchemaLocation="sites.xsd">
 <site unique-id="p1">
  <nom>Nicolas le Panda</nom>
  <position x="48.7494" y="-2.1108"/>
  <photo nom="http://www.piloteWeb.com/p1.gif" format="gif"/>
 </site>
</sites>
```

et le document XML référençant une variante du premier schéma :

```
<?xml version="1.0" encoding="UTF-8"?>
<sites xmlns:xsi="…" xsi:noNamespaceSchemaLocation="sitesVariante.xsd">
 <site unique-id="p1">
    <nom><prénom>Nicolas</prénom> le Panda</nom>
  <position x="48.7494" y="-2.1108"/>
  <photo nom="http://www.piloteWeb.com/p1.gif" format="gif"/>
 </site>
</sites>
```

Le passage à la variante revient donc, le cas échéant, à modifier physiquement les documents XML auxquels la variante s'applique. Les anciens documents peuvent cohabiter avec les nouveaux.

Schéma avec un espace de noms cible

Dans le cas où le schéma aurait un espace de noms cible, deux cas de figure se présentent selon que le schéma variant :

- est une copie du schéma initial : le nom de l'espace de noms cible peut alors être conservé ou modifié ;
- s'appuie sur les possibilités de redéfinition de XML Schema : le nom de l'espace de noms cible doit être conservé.

Le schéma variant est une copie du schéma initial

Nous distinguerons les cas d'une révision (le nouveau schéma remplace l'ancien) et d'une version (le nouveau schéma cohabite avec l'ancien).

Si la variante est une révision du schéma initial, le nom de l'espace de noms est conservé. Dans les documents XML concernés par cette révision, il faut changer l'URI du schéma et le remplacer par celui du fichier contenant le schéma variant (repère ⓿).

```
<si:sites xmlns:si="sites"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="sites sitesVariante.xsd"> ← ⓿
<si:site unique-id="ID000000">
  <si:nom><si:prénom>Nicolas</si:prénom> le Panda</si:nom>
   <si:photo nom="p1" format="gif"/>
   <si:position>
    <si:x>48.7494</si:x>
    <si:y>-2.1108</si:y>
  </si:position>
</si:site>
</si:sites>
```

Si la variante est une nouvelle version du schéma, on lui attribue un nouveau nom d'espace de noms. Dans les documents XML concernés par la nouvelle version, il faut remplacer les noms et URI de l'ancien schéma par ceux de la variante (repères ❶ et ❷).

```
<si:sites xmlns:si="sitesVariante" ← ❶
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="sitesVariante sitesVariante.xsd"> ← ❷
<si:site unique-id="ID000000">
  <si:nom><si:prénom>Nicolas</si:prénom> le Panda</si:nom>
  <si:photo nom="p1" format="gif"/>
  <si:position>
   <si:x>48.7494</si:x>
   <si:y>-2.1108</si:y>
  </si:position>
</si:site>
</si:sites>
```

Dans un tel cas de figure, la question qui ne manquera pas de se poser est celle de la mise en commun des parties semblables des modèles et documents XML associés. Cela est possible en utilisant les mécanismes d'inclusion et redéfinition de XML Schema (composants `xsd:include`, `xsd:import` et `xsd:redefine`). Nous allons voir ci-après les cas de l'inclusion et de la redéfinition.

Le schéma variant est une redéfinition du schéma initial

Avec une redéfinition, on peut réutiliser sans les copier toutes les définitions du schéma initial qui ne sont pas concernées par la variante. L'inconvénient, c'est que l'espace de noms cible de la variante est forcément égal à celui du schéma initial.

Dans l'exemple suivant, on montre un schéma initial et sa variante créée au moyen d'une redéfinition.

Le schéma initial a été rendu compatible avec les exigences de la fonction de redéfinition de XML Schema ; c'est la raison pour laquelle il est différent de celui présenté au chapitre 5. Toutefois, il ne présente pas à proprement parler une variante du schéma originel utilisé dans PiloteWeb puisqu'il définit exactement les mêmes structures et vocabulaires. Seule change la manière dont sont définis les objets.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="sites"
           xmlns:sites="sites">
<xs:complexType name="nomType" mixed="true"/>
<xs:element name="nom" type="sites:nomType"/>
<xs:complexType name="photoType">
  <xs:attribute name="nom" type="xs:anyURI" use="required"/>
  <xs:attribute name="format" type="xs:string"/>
</xs:complexType>
<xs:complexType name="siteType">
  <xs:sequence>
   <xs:element ref="sites:nom"/>
   <xs:element name="photo" type="sites:photoType" minOccurs="0"/>
   <xs:element name="position">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="x" type="xs:decimal"/>
      <xs:element name="y" type="xs:decimal"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="unique-id" type="xs:ID" use="required"/>
</xs:complexType>
<xs:element name="sites">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="site" type="sites:siteType" minOccurs="0"
                maxOccurs="unbounded"/>
```

```
    </xs:sequence>
   </xs:complexType>
 </xs:element>
 </xs:schema>
```

Le schéma variant est obtenu par redéfinition du type `nomType` du schéma initial :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="sites"
           xmlns:sites="sites">
<xs:redefine schemaLocation="sites.xsd">
<xs:complexType name="nomType" mixed="true">
  <xs:sequence minOccurs="1">
   <xs:element name="prenom" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
</xs:redefine>
</xs:schema>
```

Dans notre exemple, nous ne pourrions modifier que les définitions des types globaux. La création d'une variante par cette technique peut donc rendre obligatoire une révision du schéma initial, ce que nous avons justement fait ici.

La variante est obtenue par inclusion du schéma initial et dérivation de ses types

Dans ce cas, la variante ne permet pas de modifier les définitions des éléments du modèle initial, mais seulement d'en ajouter. Les types globaux définis dans le modèle initial peuvent quant à eux faire l'objet de dérivation par restriction ou extension, conformément aux règles édictées par XML Schema.

Dans l'exemple suivant, nous présentons le cas d'une variante créée par inclusion à partir d'un schéma initial par inclusion. Dans cette variante, le type global `nomType` est dérivé pour donner le nouveau type `newNomType`. Toutefois, ce nouveau type ne peut être substitué simplement au type associé à l'élément `nom`. Ce dernier ayant été défini globalement dans le premier schéma, il ne peut pas faire l'objet d'une redéfinition dans la variante (puisque nous opérons une simple inclusion et non une redéfinition du schéma initial). Pour introduire le nouveau type, il faut créer un nouvel élément ou utiliser l'attribut flottant `xsi:type` directement dans les documents XML. On doit se souvenir que XML Schema n'autorise la dérivation que des types et non des éléments. Le schéma initial doit donc être écrit en conséquence et proposer une séparation complète des définitions des types de celles des éléments.

Dans la variante, la technique de l'inclusion ne donne pas le choix et l'introduction d'un nouveau nom d'élément oblige tout simplement à récrire la définition de l'élément le contenant ! De proche en proche, c'est potentiellement la totalité du schéma initial qu'il faut récrire…

Le typage dynamique dans le corps des documents XML est la seule solution qui reste avec l'inclusion. Nous la donnons en exemple ci-après.

On voit que l'inclusion limite beaucoup les possibilités de création de variantes.

Le schéma initial en est :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="nomType" mixed="true"/>
<xs:element name="nom" type="nomType"/>
<xs:complexType name="photoType">
  <xs:attribute name="nom" type="xs:anyURI" use="required"/>
  <xs:attribute name="format" type="xs:string"/>
</xs:complexType>
<xs:complexType name="siteType">
  <xs:sequence>
   <xs:element ref="nom"/>
   <xs:element name="photo" type="photoType" minOccurs="0"/>
   <xs:element name="position">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="x" type="xs:decimal"/>
      <xs:element name="y" type="xs:decimal"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="unique-id" type="xs:ID" use="required"/>
</xs:complexType>
<xs:element name="sites">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="site" type="siteType" minOccurs="0"
maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Il valide le document XML suivant :

```
<sites xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="sites.xsd">
<site unique-id="ID000000">
  <nom>Nicolas le Panda</nom>
  <photo nom="photo.gif" format="gif"/>
  <position>
   <x>48.7494</x>
   <y>-2.1108</y>
  </position>
</site>
</sites>
```

Le schéma variant s'appuyant sur une inclusion du schéma initial est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:include schemaLocation="sites.xsd"/>
<xs:complexType name="newNomType" mixed="true">
  <xs:complexContent>
   <xs:extension base="nomType">
    <xs:sequence minOccurs="1">
     <xs:element name="prenom" type="xs:string"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
</xs:complexType>
</xs:schema>
```

Il valide le document XML suivant, dans lequel on peut observer l'utilisation de l'attribut `xsi:type` :

```
<sites xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="sitesVarianteInclusion.xsd">
<site unique-id="ID000000">
  <nom xsi:type="newNomType"><prenom>Nicolas</prenom> le Panda</nom>
  <photo nom="photo.gif" format="gif"/>
  <position>
   <x>48.7494</x>
   <y>-2.1108</y>
  </position>
</site>
</sites>
```

Conséquence de la création d'une variante d'un sous-schéma

> **VOCABULAIRE Sous-schéma**
>
> Dans ce livre, nous appelons sous-schéma un schéma XML conçu pour être inclus dans un autre. On parle donc de sous-schéma comme on parlerait d'un sous-document. Cependant, il convient de noter que XML Schema ne définit aucunement la notion de « sous-schéma », tout simplement parce qu'il ne s'y attache rien de spécifique. Comme tout schéma, un sous-schéma doit être valide indépendamment des schémas qui l'utilisent, ou que lui-même utilise.

La création d'une variante d'un sous-schéma peut avoir des conséquences sur les schémas qui l'utilisent.

Comme nous l'avons vu plus haut, la modification du sous-schéma peut entraîner :

- la création d'un nouveau fichier contenant le nouveau sous-schéma ;
- la création d'un nouvel espace de noms cible.

Dans l'un ou l'autre de ces cas, les schémas principaux devront être modifiés. Si ces derniers sont eux-mêmes inclus dans d'autres, il se pourrait, par effet boule de neige, que nous soyons amenés à toucher un grand nombre de schémas.

Un sérieux problème de conception de schéma se pose alors potentiellement : en effet, pour éviter de prendre le risque de doubler toutes les définitions de tous les éléments et attributs, la seule solution consiste à créer un ou plusieurs catalogue(s) de types. Pour que cette conception des schémas soit cohérente avec XML Schema, il faut impérativement que les types soient définis indépendamment des éléments et des attributs, car XML Schema permet seulement de dériver des types et non des définitions d'éléments ou d'attributs.

Quand notre conception nous amène à créer une pyramide constituée de schémas et de sous-schémas, il faut commencer par créer les catalogues de types qui constitueront la base de cette pyramide. En clair, cela signifie que nous devons créer des schémas de bas niveau dans lesquels ne seront définis que des types, et plus nous irons vers le sommet de la pyramide et plus nos schémas définiront des structures complexes.

Les schémas intermédiaires seront constitués des définitions d'attributs, des éléments simples et globaux.

Cela revient finalement à parfaitement isoler les types des structures, ce qui rationalise énormément la programmation des schémas XML.

Conséquence des variantes sur les liens

Quand une structure de données XML est commune à plusieurs schémas, il est intéressant qu'elle soit directement associée à un document. Dans ce cas, le document contient des données parfaitement conformes à cette seule structure. On suppose que les données sont ainsi factorisées entre tous les documents XML utilisant cette structure commune.

Cela a été fait dans l'application PiloteWeb qui sert de trame aux exemples de cet ouvrage. Les structures `sites` et `partenaires` sont partagées avec les autres selon l'organisation hiérarchique des modèles que nous présentons à la figure 6-1. Attention, la figure présente bien une hiérarchie de dépendances entre modèles et non une hiérarchie d'éléments XML !

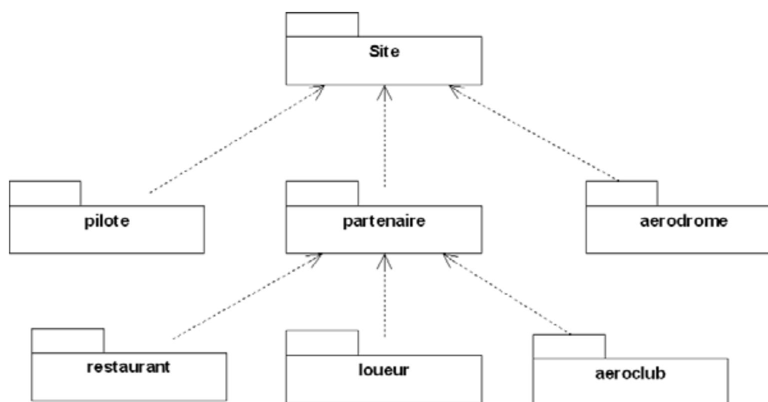


Figure 6–1
La hiérarchie des modèles de notre application

Une telle représentation de la cascade des schémas utilisés dans PiloteWeb, ou architecture du modèle, met en évidence non seulement l'intérêt des sous-schémas, mais également les dépendances que cela engendre, notamment dans l'hypothèse de création de variantes.

Si une variante du modèle `pilotes` est créée, l'impact sera circonscrit au seul document XML correspondant, `pilotes.xml`. Si une variante de `partenaires` est créée, l'effet se fera sentir sur trois autres modèles : `restaurants`, `loueurs` et `aéroclubs`. Enfin, si une variante de `sites` est créée, tous les modèles de PiloteWeb seront peut-être concernés.

Cela dit, nous avons choisi dans PiloteWeb de faire correspondre un document XML par schéma du modèle. Les données sont mises en commun grâce à l'utilisation de XLink dans les documents XML ; nous donnons ci-après en exemple la relation qui unit ainsi les données d'un site à celles de la structure pilote.

Voici un exemple de structure `site` :

```
<?xml version="1.0" encoding="UTF-8"?>
<sites xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="sites.xsd">
<site unique-id="i15">
<nom>La bonne étoile</nom>
<photo nom="labonneetoile.png" format="png"/>
<position>
<x>47.3105</x>
<y>2.1566</y>
</position>
</site>
</sites>
```

Et voici la structure `pilote` correspondante :

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml version="1.0" encoding="UTF-8"?>
<pilotes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="pilotes.xsd">
<pilote unique-id="i19" publie="1">
<link xlink:type="simple" xlink:show="embed" xlink:href="sites.xml/
#xpointer(descendant::site[@unique-id='i15'])"/> ← ❸
<email>snail@piloteweb.com</email>
<pwd>secret</pwd>
</pilote>
```

Pour conserver l'indépendance des données (que l'on a voulu stocker dans des fichiers séparés), nous avons utilisé un élément `link` porteur des attributs standards de la recommandation du W3C XLink. Les données de la structure `site` *ad hoc* sont virtuellement intégrées (`xlink:show="embed"`) à la structure `pilote` via le lien ❸ qui référence l'identifiant (en l'occurrence la valeur `i15`) de la structure `site` souhaitée.

Ainsi avons-nous modélisé une hiérarchie de modèles, mis en évidence les structures de données communes, tout en échappant au problème des conséquences de la création de variante. Avec la technique de modélisation que nous présentons ici, toute création de variante a de fortes chances d'être circonscrite à la seule variante.

En résumé…

Dans ce chapitre, nous avons vu, cas pratiques à l'appui, les différentes possibilités offertes pour créer et gérer des variantes.

Nous avons également passé en revue les raisons qui pouvaient amener à la création de variantes, ainsi que les conséquences de ces créations.

XML Schema est très rigide dans les possibilités qu'il donne de création de variantes. Il se trouve que cela profite à la conception des architectures de modèles car l'expérience montre que la pyramide de schémas qu'impose XML Schema est salutaire. Elle permet tout à la fois d'obtenir des modèles génériques plus propres et d'avoir un meilleur contrôle des variantes.

Enfin, nous avons vu que mettre en œuvre XLink permettait de s'affranchir presque totalement des problèmes de gestion des variantes. Grâce à la solution que nous présentons ici, les modèles ainsi que les documents XML restent indépendants les uns des autres. Les évolutions sont donc prises en charge plus simplement.

Après avoir organisé nos modèles sous la forme d'une pyramide de schémas, nous allons voir dans le chapitre suivant comment organiser les différentes couches de programmation.

7

Étape 7 - Organiser les différentes couches de programmation

Nous allons traiter principalement de la place de XML dans les applications web, mais il ne faut pas croire pour autant qu'il n'a trait qu'à ce type d'applications ; toutes sont concernées par XML !

C'est en rationalisant les couches de traitement qu'on améliore la qualité des programmes. Cela concerne aussi bien les applications de commerce électronique que la production de documents ou encore l'intégration d'applications.

Comme dans le reste de cet ouvrage, nous ne nous intéressons dans ce chapitre qu'aux seuls programmes de traitement relatifs à XML.

Nous utilisons dans ce chapitre l'exemple de PiloteWeb, notre application web qui s'appuie sur XML de bout en bout : depuis le stockage des données dans une base de données XML jusqu'à l'affichage des pages dans le navigateur en passant par l'utilisation de formulaires de saisie et un service web. Notre application repose sur un serveur de pages JSP, des API Java et l'utilisation de XQuery, Xpath, DOM, XLink et XSLT.

Nous allons montrer comment le développement d'une telle application doit faire alterner la modélisation des schémas XML de stockage avec la prise en compte des couches et langages de programmation.

Présentation générale

Contrairement à ce que l'on imagine trop vite, une application web fondée sur XML ne se ramène pas à un problème de transformation du XML en HTML au moyen de feuilles de style XSLT. La figure 7-1 schématise une telle approche. Ce n'est qu'une vue simpliste, et donc fausse, d'une application web utilisant des données XML.

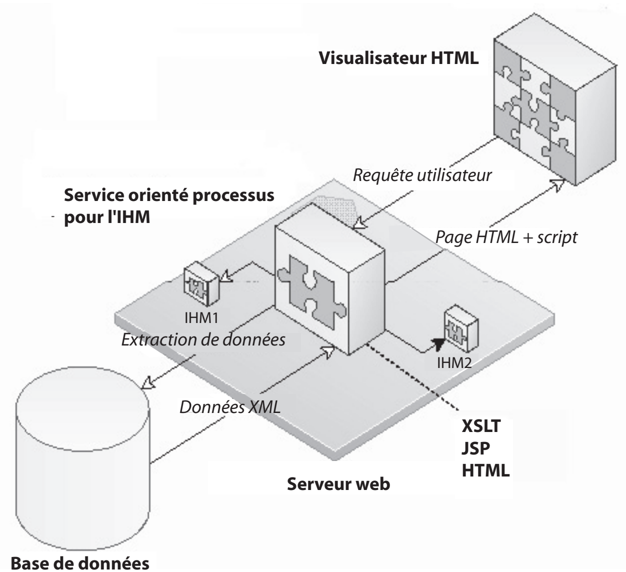


Figure 7–1
Représentation trop simpliste d'une application web utilisant XML

La figure 7-1 montre un flot XML, produit par la conversion d'un format quelconque ou fruit d'une requête XQuery auprès d'une base de données XML. Ces données XML sont affichées dans un navigateur par transformation en HTML au moyen d'une feuille de styles XSLT. Un tel schéma ne fonctionne pas, car il présente les inconvénients suivants :

- Il n'autorise, au maximum, que deux étapes de transformation des données XML : au moment de l'exécution de la requête XQuery et au moment de la transformation en HTML par XSLT. C'est insuffisant.
- Il ne tient pas compte du rôle du serveur web et de son architecture propre.
- Il ne représente pas la partie transactionnelle des accès à la base de données ni le dialogue provoquant l'exécution des requêtes XQuery.
- Il ne dit rien quant à la gestion des messages d'erreur.
- Il ne permet pas de tenir compte des transformations qui seraient conditionnées par des contraintes externes, pourtant souvent nécessaires.

C'est la prise en compte de ces points qui rend si difficile la conception des applications, dont la réalité même est bien complexe et malaisée à exposer et représenter visuellement. Cela est probablement d'autant plus vrai que les applications d'Internet sont si nombreuses que n'importe quelle représentation ou tentative de modélisation pourrait être discutée pendant des heures. Afin de cerner notre propos, nous nous basons sur un cas d'école simple et didactique, mais dont nous pensons qu'il peut toutefois servir de modèle à des cas plus sophistiqués.

La force du modèle que nous présentons réside dans sa cohérence avec la méthode d'analyse présentée à l'étape 1 : cette dernière aura permis d'identifier les différentes étapes de fabrication de l'information. Le modèle en résultant peut servir de base de travail pour le découpage logique des traitements.

Dans une application web, on ne voit du monde des documents que des pages HTML qui peuvent elles-mêmes contenir des programmes exécutés au moment de l'affichage. On ne peut plus dès lors se contenter de représenter les documents structurés sous la forme du classique triptyque fond, forme, structure, comme le montre la figure 7-2. Il faut lui ajouter un quatrième volet, celui des programmes applicatifs embarqués dans le document, ce que nous représentons par un losange à la figure 7-3.

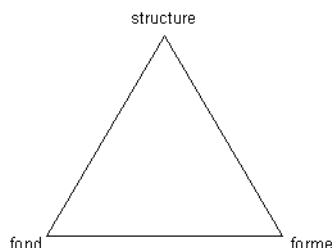


Figure 7–2
Le classique triptyque fond, forme, structure, représentant un document structuré

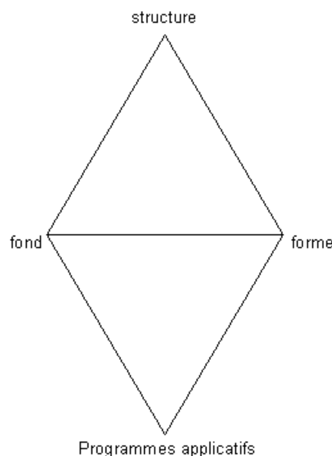


Figure 7–3
Les quatre dimensions d'un document XML applicatif

La figure 7-1 est erronée précisément parce que cette dimension spécifique aux documents web est absente. Nous devons y ajouter les éléments qui traduisent les capacités d'intégration : au minimum, il s'agit d'un serveur d'application web, d'un langage de gestion des requêtes XQuery, et enfin d'un langage de génération de pages.

Un meilleur modèle de base pour notre conception d'application est celui représenté à la figure 7-4.

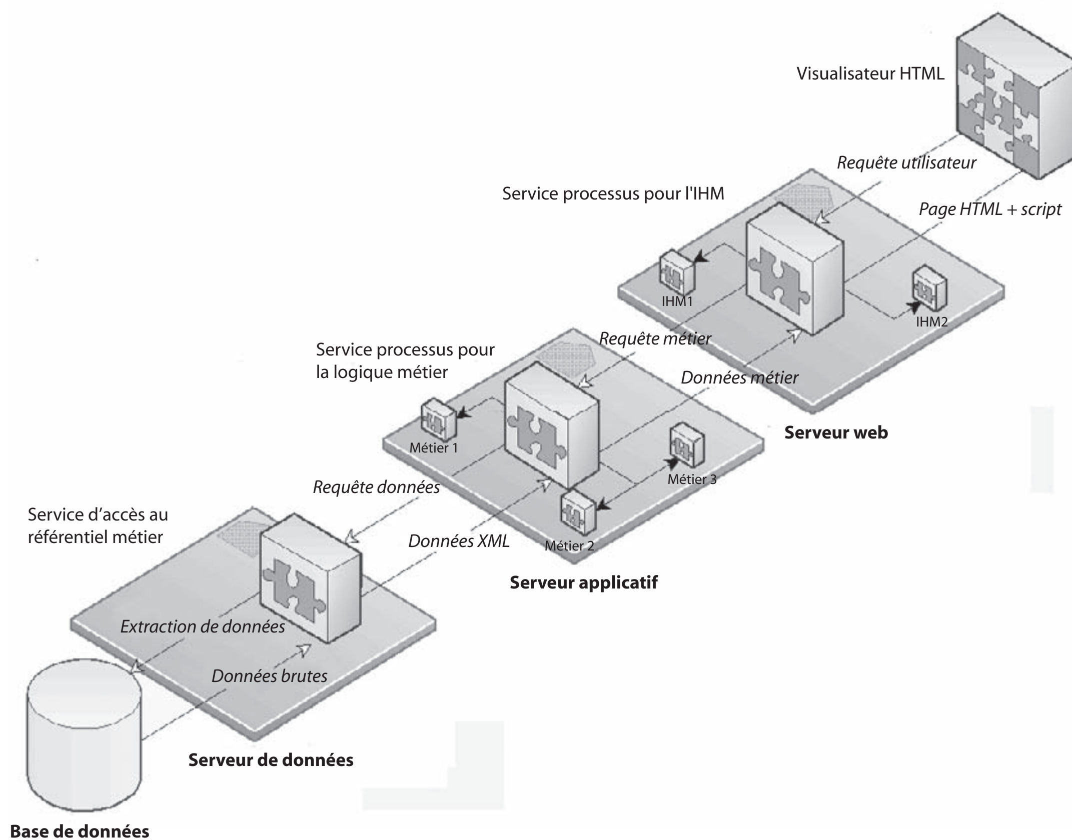


Figure 7–4 Représentation plus exacte d'une architecture logique applicative

La seule intelligence applicative qu'on puisse ajouter dans un document « simplement » HTML se ramène à des programmes embarqués dans le code, tel du JavaScript. Avec des documents XML embarquant une logique applicative via leurs balises, on passe par des appels de pages dont l'exécution produit des documents XML qui, après conversion par XSLT, produisent des documents HTML qui, à leur tour, peuvent contenir des programmes exécutés au moment de l'affichage.

C'est ce dernier cycle de traitements que représente la figure 7-4. Partons d'une requête émise depuis un visualiseur HTML et suivons le parcours de cette requête jusqu'à l'affichage de la réponse qu'elle provoque :

1. La requête est reçue par un serveur d'applications web (*service des processus d'interfaces*) ; elle consiste en une demande d'activation de page. Le serveur d'applications web décortique la page appelée.
2. Il exécute, via les arguments de la requête, le programme demandé (*service des processus métier*) dont l'appel est codé dans la page.
3. Ce programme peut exécuter un simple calcul ou être lui-même l'auteur d'une requête vers un *référentiel métier* (une base de données par exemple).
4. Ce dernier service va finalement exécuter la requête physique qui consiste à récupérer les données demandées et les retourner au programme appelant.
5. Le programme appelant (le service des processus métier) va transmettre les données au service de processus d'interfaces après les avoir éventuellement transformé les données. Cet envoi peut être un flot de données XML ou HTML.
6. Enfin, le service des processus d'interfaces va convertir les données en HTML définitif à des fins d'affichage dans le visualiseur d'où est partie la requête initiale.

Dans J2EE, le langage de génération de pages est JSP. Nous l'utilisons pour écrire les squelettes de pages web. Le langage correspondant au serveur de processus métier est Java sous la forme d'un *bean*. La technique des beans Java, reconnue des serveurs d'application J2EE, permet en particulier de garantir le partage et la persistance des données sur le serveur entre plusieurs appels de pages JSP. Enfin, quand on dispose d'une base de données XML, le langage de requête utilisé par le service du référentiel métier est XQuery.

Le graphique de la figure 7-5 met en évidence l'ensemble des langages utilisés depuis l'appel initial d'une page JSP jusqu'à l'affichage de la réponse sous la forme d'une page HTML. Nous allons continuer à étudier ces langages et la manière dont ils peuvent s'imbriquer les uns dans les autres. La figure 7-6 donne enfin une représentation épurée de l'enchaînement des langages : dans cette représentation, toute autre notion a été retirée.

On trouve à l'extrême gauche de la figure 7-6 les données XML stockées, souvent selon un schéma spécifique au stockage, et à l'extrême droite les pages HTML affichées. Si le rôle des schémas XML est important dans la partie gauche, on ne peut pas en dire autant des pages affichées. HTML est un langage de présentation qui ne traite que de la mise en forme des données, toute notion de structure ayant donc disparu des données.

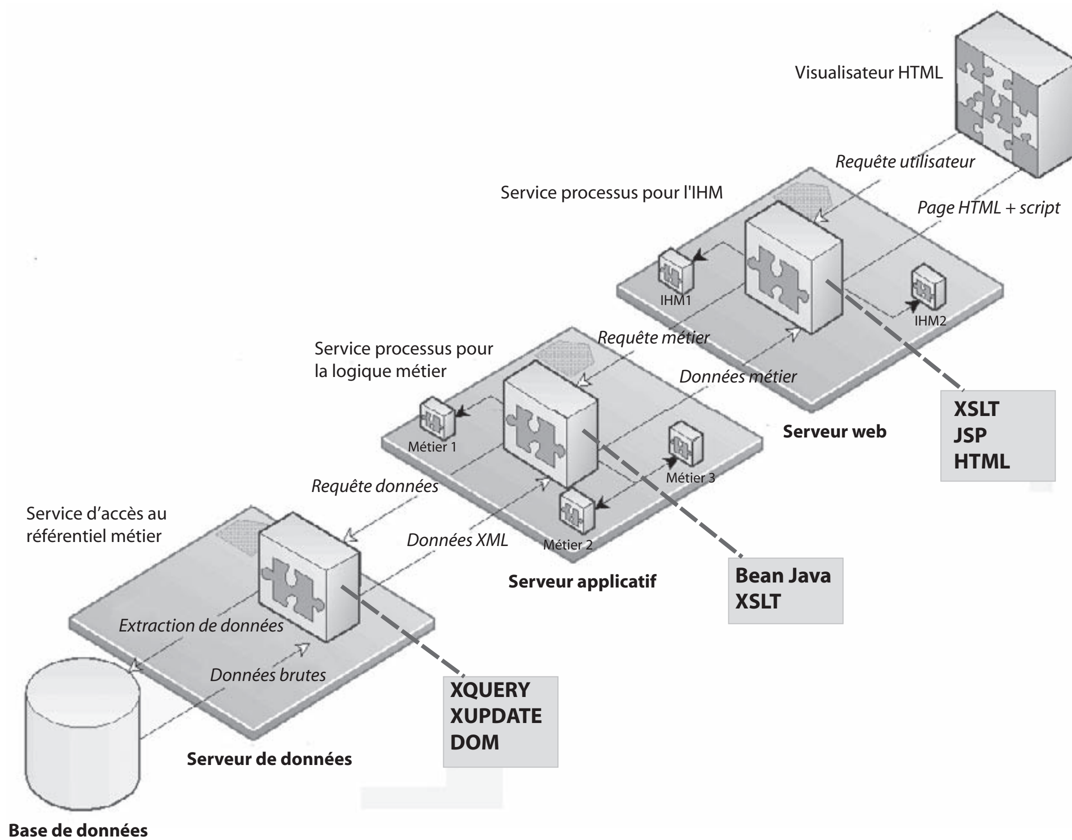


Figure 7–5 Représentation des langages de programmation associés à l'architecture logique applicative

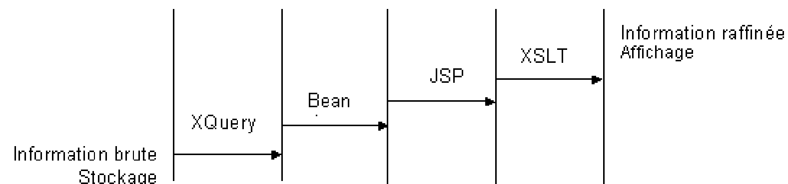


Figure 7–6
Schéma des différentes couches de programmation

La figure 7-6 est plus apte à nous faire réfléchir sur la rationalisation des différentes couches de programmation que les figures 7-1 et 7-4, car elle met en évidence les problèmes de modélisation orientée métier (à gauche) et orientée présentation (à droite). Les langages et les étapes de transformation ne sont alors qu'autant de processus visant à transformer un contenu structuré (l'information brute métier) en une forme lisible (information raffinée consommable). Cette manière de voir les choses

est intéressante pour la modélisation globale du système. Dans la section suivante, nous montrons comment cette représentation de l'architecture du système amène une meilleure méthode de conception et d'analyse des modèles XML du système.

D'un modèle de stockage à un modèle de présentation

Les transformations dont nous avons parlé jusqu'à maintenant ne sont que la partie visible d'un problème méthodologique : comment définir avec certitude et à l'avance (dès les phases de conception) quelle partie de la programmation prendra en charge telle partie des traitements ?

Si l'on accepte l'idée que les données passent par six états différents et pas moins de quatre types de transformations avant d'être affichées dans un quelconque navigateur, alors les questions relatives à la rationalisation des traitements sont les suivantes :

- Quel langage utiliser pour un type donné de transformation ?
- À quel stade une transformation doit-elle être faite ?
- Comment écrire les spécifications correspondantes ?
- Comment rendre génériques, communs, les programmes de transformation ?

Répondre à ces questions est d'autant plus malaisé que les transformations élémentaires sont nombreuses. Il est alors difficile de toutes les modéliser et documenter.

Pour y parvenir toutefois, nous allons scinder le problème en deux volets distincts et considérer :

1. d'une part, que toute transformation n'a comme seul objectif que de faire passer les données d'un modèle A à un modèle B ;
2. d'autre part, que les langages imposent plus ou moins d'eux-mêmes l'ordre des opérations de transformation.

Ces deux points seront détaillés dans les deux prochaines sections.

Organisation du modèle de stockage

Pour illustrer ce point, nous allons utiliser comme exemple l'un des cas traités dans notre application école PiloteWeb : le traitement des coordonnées géophysiques (latitude/longitude) des aérodromes, pilotes et partenaires (restaurants, loueurs de voitures…). En effet, PiloteWeb calcule, pour chaque aérodrome affiché, les pilotes et prestataires de services proches de l'aérodrome en question. Cela se fait par comparaison de leurs positions géographiques respectives.

L'application PiloteWeb est conforme au schéma présenté plus haut à la figure 7-5. Les données sont exprimées en XML d'un bout à l'autre de l'application et ne sont transformées en HTML qu'au tout dernier moment. Cette dernière transformation est assurée par une feuille de styles XSLT, tandis que toutes les autres sont calculées par des programmes Java ou JSP.

Dans la base XML utilisée, les données sont organisées autour de sept schémas et dépendent les unes des autres conformément à la figure 7-7.

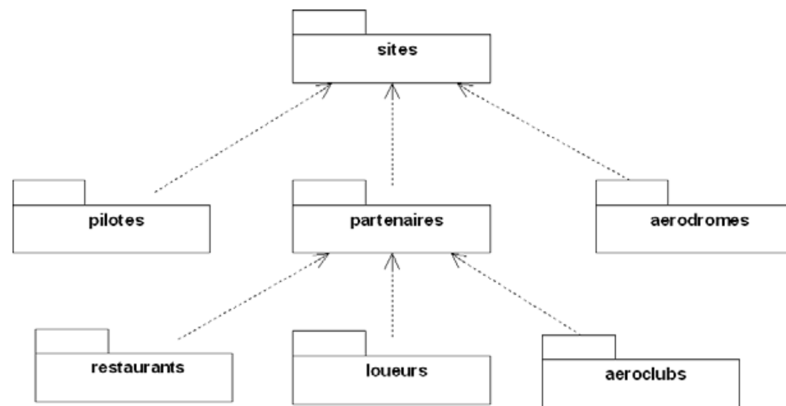


Figure 7–7
Organisation des schémas de PiloteWeb

Comme nous l'avons expliqué lors de l'étape 6, ce modèle hiérarchique de schémas XML exprime le fait que certaines données sont partagées par plusieurs objets. C'est ainsi que le schéma XML `sites` contient la structure qui permet d'exprimer la position géographique de n'importe quel objet de la base : `pilotes`, `aerodromes`, `partenaires`, `restaurants`, `loueurs` et `aeroclubs`. Les trois derniers (`restaurants`, `loueurs` et `aeroclubs`) ont eux-mêmes une structure en commun exprimée par le schéma XML `partenaires`, qui décrit les informations pratiques communes à tous les partenaires : nom, numéro de téléphone, disponibilité, etc.

L'organisation présentée à la figure 7-7 montre les relations de dépendance entre les schémas XML : ceux du bas héritent de ceux du haut.

Les données sont stockées dans autant de documents XML qu'il y a de schémas dans cette organisation hiérarchique, soit sept au total. Il faut donc noter ici la très grande différence qu'il y a entre stocker toutes les données d'un même objet dans un seul document XML et les répartir dans plusieurs documents XML différents (comme c'est le cas ici). Par exemple, les données descriptives d'un loueur de voiture sont, dans notre cas d'école, stockées dans trois documents XML : l'un stocke la position géographique, l'autre les informations pratiques et le dernier les données spécifiques.

Pour obtenir la totalité des informations relatives à un loueur de voiture en particulier, l'application devra faire la jointure de ces trois documents XML.

Le modèle `sites` est commun à tous les autres. Les schémas `pilotes` et `aérodromes` héritent de ce modèle, qu'ils complètent par leurs propres structures XML car les pilotes et les aérodromes ont chacun des caractéristiques spécifiques. Aucun autre schéma n'hérite de ces deux modèles : ils sont terminaux. Il n'en va pas de même pour `partenaire` qui, comme nous l'avons vu, sert de base à trois autres modèles de PiloteWeb : `restaurants`, `loueurs` et `aeroclubs`.

Nous allons étudier deux de ces schémas dans la suite de ce chapitre et montrer des exemples valides des documents XML correspondants.

Schémas sites et pilotes

Tous les objets que notre application manipule ont en commun un repérage géographique, composé d'un nom, d'une position géographique et d'une photo optionnelle. Ces données font l'objet du schéma `site`.

L'élément racine est `sites` (au pluriel). Il contient autant d'éléments `site` (au singulier) que d'objets introduits dans la base. En voici le schéma :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
targetNamespace="sites"
xmlns:sites=" sites">
 <xs:element name="nom" type="xs:string"/>
 <xs:complexType name="photoType">
  <xs:attribute name="nom" type="xs:anyURI" use="required"/>
  <xs:attribute name="format" type="xs:string"/>
 </xs:complexType>
 <xs:complexType name="positionType">
  <xs:sequence>
   <xs:element ref="sites:x"/>
   <xs:element ref="sites:y"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="siteType">
  <xs:sequence>
   <xs:element ref="sites:nom"/>
   <xs:element name="photo" type="sites:photoType" minOccurs="0"/>
   <xs:element name="position" type="sites:positionType"/>
  </xs:sequence>
  <xs:attribute name="unique-id" type="xs:ID" use="required"/>
 </xs:complexType>
```

```
 <xs:element name="sites">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="site" type="sites:siteType"
                minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="x" type="xs:decimal"/>
 <xs:element name="y" type="xs:decimal"/>
</xs:schema>
```

L'attribut `unique-id` de l'élément `entite` identifie chaque élément `site` de manière unique afin d'y faire référence à partir des instances des modèles enfants.

Un document XML conforme à ce schéma se codifie ainsi :

```
<?xml version="1.0" encoding="UTF-8"?>
<sites:sites xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="sites sites.xsd" xmlns:sites="sites">
  <sites:site unique-id="i1">
    <nom>Pilote de Démo</nom>
    <photo nom="pingouin.jpg" format="jpg"/>
    <position>
      <x>48.7494</x>
      <y>-2.1108</y>
    </position>
  </sites:site>
  <sites:site unique-id="i2">
    <nom>Nicolas le Panda</nom>
    <photo nom="merlin.gif" format="gif"/>
    <position>
      <x>47.9505</x>
      <y>-0.19</y>
    </position>
  </sites:site>
  <sites:site unique-id="i3">
    <nom>Max</nom>
    <photo nom="max.gif" format="gif"/>
    <position>
      <x>47.9347</x>
      <y>-0.2172</y>
    </position>
</sites>
```

S'agissant du schéma `pilotes`, étudions tout d'abord l'instance XML des pilotes :

```
<?xml version="1.0" encoding="UTF-8"?>
<pilotes:pilotes xmlns:pilotes="www.piloteWeb.xml/pilotes"
          xmlns:xlink="http://www.w3.org/1999/xlink"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="www.piloteWeb.xml/pilotes pilotes.xsd">
  <pilote unique-id="i16" publie="1">
    <link xlink:type="simple" xlink:show="embed"
          xlink:href="sites.xml/#xpointer(
             ➥descendant::site[@unique-id='i1'])"/> ← ❶
    <email>pilote@piloteweb.com</email>
    <pwd>avion</pwd>
  </pilote>
  <pilote unique-id="i17" publie="1">
    <link xlink:type="simple" xlink:show="embed"
          xlink:href="sites.xml/#xpointer(
             ➥descendant::site[@unique-id='i2'])"/> ← ❶
    <email>panda@piloteweb.com</email>
    <pwd>secret</pwd>
  </pilote>
</pilotes:pilotes>
```

Les pilotes sont caractérisés par un nom, une photo et une position géographique, données communes à tout objet géré par PiloteWeb. Ces dernières sont ici référencées par le biais d'un lien vers le document XML `sites` (repère ❶). Les pilotes ont en propre une adresse électronique (élément `email`) et un mot de passe (élément `pwd`).

Voici le schéma correspondant :

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="piloteType">
  <xs:sequence>
   <xs:element name="link">
    <xs:complexType>
     <xs:anyAttribute namespace="http://www.w3.org/1999/xlink"
                      processContents="strict"/>
    </xs:complexType>
   </xs:element>
   <xs:element name="email" type="emailType" nillable="false"/>
   <xs:element name="pwd" type="xs:NCName" nillable="false"/>
  </xs:sequence>
  <xs:attribute name="unique-id" type="xs:ID"/>
  <xs:attribute name="publie" type="xs:boolean" default="1"/>
 </xs:complexType>
```

```
 <xs:simpleType name="emailType">
  <xs:restriction base="xs:string">
   <xs:pattern value="(\c)*@(\c)*"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:element name="pilote" type="piloteType"/>
 <xs:element name="pilotes">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="pilote" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

L'organisation des schémas présentée à la figure 7-7 se traduit par la mise en œuvre de liens XLink. Comme nous l'avons précisé à l'étape 6, nous recommandons l'usage de XLink pour minimiser les efforts de migration lors d'une éventuelle évolution des modèles.

Les modèles sont ici suffisants pour comprendre la nature des données exprimées. Leur seule lecture apporte déjà beaucoup d'informations sur le type d'application, et ce même si la totalité de la sémantique n'est pas exprimée dans le schéma (telles les règles syntaxiques et la signification exacte de l'adresse `email`).

Par contraste, nous allons observer les modèles utilisés au niveau de l'affichage de ces données. Il s'agit des DTD dites d'affichage.

> **REMARQUE Utilisation des DTD et des schémas**
>
> Dans ce chapitre, nous utilisons les schémas XML pour les modèles de stockage et les DTD pour les modèles d'affichage. Cela montre d'une part que les deux peuvent cohabiter au sein d'une même application, et d'autre part que le passage de l'un à l'autre sera probablement chose courante dans la documentation des applications de XML. En effet, l'écriture des modèles XML sous la forme de DTD est beaucoup plus concise et simple à lire que sous la forme de schémas XML, ce que nous avons mis à profit ici.

Organisation des modèles d'affichage

Les DTD dont nous allons parler dans cette section sont celles utilisées juste avant conversion en HTML des données XML ; la DTD finale HTML n'ayant, quant à elle, rien de particulier, sinon son caractère très généraliste que tout un chacun connaît.

Les DTD d'affichage représentent la somme de toutes les données accumulées au cours des différentes étapes de traitement : on va donc y retrouver les modèles des données extraites de la base et conservées jusqu'à l'affichage, de celles produites par

l'exécution des pages JSP, reçues de services web (ce qui est le cas dans PiloteWeb), et enfin résultant des calculs faits en Java.

La figure 7-8 rend compte de l'organisation des DTD utilisées dans le service des processus d'interfaces de notre application.

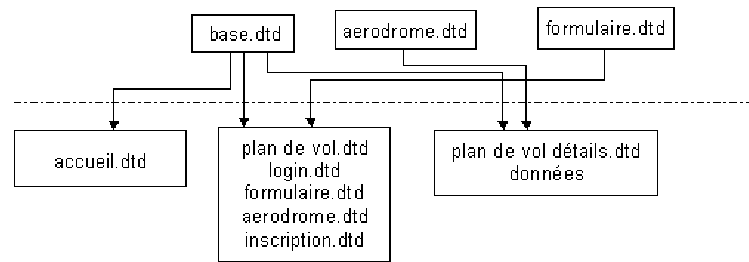


Figure 7–8
L'organisation des DTD d'affichage

Les modèles d'affichage sont divisés en deux mondes. L'un est constitué de modèles génériques indépendants les uns des autres (`base.dtd`, `aerodrome.dtd` et `formulaire.dtd`), l'autre renferme des modèles s'appuyant sur des combinaisons de ces modèles génériques.

Il y a le plus souvent autant de DTD d'affichage que de modèles de pages web à afficher, chaque page web étant la somme de plusieurs de ces DTD (par exemple, un par cadre de page web). Comme pour les schémas XML de stockage, ces DTD sont en partie dépendantes les unes des autres. Les modèles communs correspondent alors ici aux mises en page types communes à plusieurs pages web (par exemple, un cadre utilisé à l'identique dans plusieurs pages web).

Dans le cas de notre application école, nous n'allons étudier que la DTD d'affichage `formulaires.dtd` :

```
<!ENTITY % principal (p*,formulaire?)>
<!ENTITY % base SYSTEM "base.dtd">
<!ENTITY % formulaires SYSTEM "formulaires.dtd">
%base;
%formulaires;
```

Elle est la combinaison de deux DTD de base, `base.dtd` et `formulaires.dtd`.

Voici la DTD `base.dtd` :

```
<!ENTITY % principal "EMPTY">
<!ENTITY % page
"((quitter,raccourcis,principal)|(login,raccourcis,principal))">
```

```
<!ELEMENT page %page;>
<!ELEMENT quitter EMPTY>
<!ELEMENT login EMPTY>
<!ELEMENT raccourcis (a*)>
<!ELEMENT a (#PCDATA)>
<!ATTLIST a href CDATA #REQUIRED>
<!ELEMENT principal %principal;>
<!ATTLIST principal titre CDATA #REQUIRED>
```

Et la DTD `formulaires.dtd` :

```
<!ELEMENT formulaire (liste|choix|texte|passe|position|fichier)*>
<!ELEMENT liste (option+)>
<!ELEMENT option (#PCDATA)>
<!ELEMENT choix EMPTY>
<!ELEMENT texte EMPTY>
<!ELEMENT passe EMPTY>
....Suivi d'une liste de définition d'attributs.
```

La lecture de ces DTD montre immédiatement que le balisage sémantique des schémas XML de stockage a laissé la place à un balisage de composition sans pourtant être déjà le balisage HTML final (trop indépendant de toute application). Ce modèle est spécifique à notre application dont il traduit la cohérence des présentations des pages web.

En particulier, on y trouve les éléments `login`, `quitter` et `principal` qui correspondent aux écrans de connexion à l'application, sortie et menus principaux. On comprend mieux ici pourquoi ces éléments n'ont rien à faire dans la base de données mais font partie du dialogue de l'application : ils doivent être introduits dans le flot des données à une certaine étape des traitements.

En synthèse

Dans cette section, nous avons mis en évidence l'existence de modèles qui ont pour vocation d'assurer, d'un côté le partage des données entre le plus grand nombre d'applications possibles, et de l'autre la cohérence des pages web d'une application donnée.

La transformation des données d'un modèle à un autre doit donc être étudiée comme étant la somme de micro-transformations permettant de passer d'un modèle métier (stockage des données) à un modèle applicatif (le modèle d'affichage), chacun d'eux étant en fait composé d'un ensemble de DTD organisées comme nous l'avons vu aux figures 7-7 et 7-8. Ces DTD doivent bien sûr être en cohérence avec l'architecture de l'application qui les gouverne.

Les différentes couches de programmation

Dans cette section, nous présentons les langages utilisés pour faire évoluer les données d'un modèle métier à un modèle de restitution, d'affichage et de dialogue interactif avec l'utilisateur. Nous allons voir que chaque langage ayant ses propres spécifications, toute transformation n'est pas systématiquement possible à l'endroit souhaité.

Les cinq langages de programmation utilisés dans PiloteWeb correspondent à cinq couches qui, selon la théorie, sont les suivantes :

- la mise en correspondance fonctionnelle, ou *mapping*, réalisée en XQuery ;
- la programmation en Java des objets d'entreprise ;
- la programmation des objets métier en JSP ;
- la programmation de la présentation en XSLT ;
- la programmation de l'affichage avec HTML.

Le schéma 7-9 rappelle ici ceux des figures 7-5 et 7-6 du début du chapitre. Il correspond à la chaîne logique des traitements, que l'on peut résumer ainsi : Stockage←→Référentiel Entreprise←→Processus Métier←→Processus IHM.

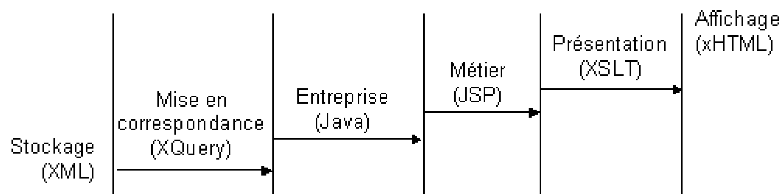


Figure 7–9
L'organisation des couches de programmation

Chaque couche fait l'objet d'une description plus précise dans les sections qui suivent.

Mise en correspondance fonctionnelle avec XQuery

Cette couche a pour mission d'apporter les données à tout programme des couches supérieures, principalement les programmes des couches entreprise et métier.

L'utilisation de requêtes XQuery peut être en effet rendue nécessaire pour transformer des données destinées tant à l'ensemble de l'entreprise qu'à une application en particulier ; il s'agit alors d'objets métier.

Par exemple, dans PiloteWeb, les positions géographiques des objets sont saisies et affichées sous la forme degré, minute, seconde (unité que l'on notera *dms* dans la suite de cette section). Nous avons pourtant choisi de les stocker sous la forme décimale, car le type décimal existe en XML Schema, alors qu'il n'existe pas de type *dms*. Cela entraîne des conversions systématiques de ces données lors de toute opération de stockage/déstockage.

Les figures 7-10 et 7-11 montrent la manière dont les positions géographiques des objets sont présentées aux utilisateurs de l'application, tandis que la figure 7-12 explique comment ces mêmes données sont stockées dans la base.

Figure 7–10
Affichage des coordonnées géographiques dans PiloteWeb

Figure 7–11
Saisie des coordonnées géographiques dans PiloteWeb

Figure 7–12
Stockage des coordonnées géographiques dans PiloteWeb

La lecture des nombres x et y stockés sous forme décimale dans la base fait l'objet d'un programme XQuery particulier. Leur conversion en *dms* ne peut être effectuée que par un programme Java acceptant comme argument d'entrée une chaîne de caractères de type décimal et retournant en sortie sa correspondance en *dms*. Ce programme n'est pas spécifique à l'application PiloteWeb. C'est un programme générique qui pourrait être utilisé par plusieurs applications différentes, un programme de type entreprise.

APARTÉ DMS

Acronyme représentant l'unité degré-minute-seconde.

Voici le code source de la requête XQuery qui retourne la position X d'un objet dont nous connaissons l'identifiant `id` (dans notre exemple, on a donné à l'identifiant la valeur 16) :

```
for $i in document('/piloteWeb-data/')//*:*[@unique-id="i16"] ← ⓿
   let $a := '/piloteWeb-data/', ← ❶
        $b := substring-before(string($i/*:link/@*:href),'/'),
        $c := concat($a,$b),
        $d := substring-after(string($i/*:link/@*:href),"unique-id='"),
        $e :=substring-before($d,"'")
for $obj in document($c)//*[@unique-id=$e]
        return <return>{$obj/position/x/text()}</return> ← ❷
```

Cette requête ne va pas chercher directement la valeur X correspondant à l'objet `i16` car, dans PiloteWeb, les coordonnées `(x,y)` d'un objet sont stockées dans un document XML distinct de l'objet lui-même. La requête XQuery recherche donc l'objet dont on connaît l'identifiant (repère ⓿), puis décortique le lien qui relie cet objet au document XML contenant les coordonnées `(x,y)`. C'est le but de la série des ordres `let` qui commence au repère ❶. Enfin, ce programme retourne la coordonnée X trouvée (repère ❷).

La liste déroulante des aérodromes utilisée dans les pages web de l'application est pour partie indissociable du code des pages en question. Ces programmes se trouvent dans la couche métier. Le calcul de l'affichage de cette liste se divise en deux niveaux de programmation; l'un du niveau métier, l'autre du niveau mise en correspondance :

- niveau métier : le code JSP qui récupère cette liste et la transforme en menu déroulant ;
- niveau entreprise : le programme Java qui récupère la liste d'aérodromes retournée par la requête XQuery sous forme de DOM ;
- niveau mise en correspondance : la requête XQuery qui ramène la liste des aérodromes.

La requête XQuery utilisée est la suivante :

```
for $i in document('/piloteWeb-data/')/*:aerodromes/*:aerodrome ← ❸
      let $unique-id := $i/@unique-id, ← ❹
           $a := '/piloteWeb-data/',
           $b := substring-before(string($i/*:link/@*:href),'/'),
           $c := concat($a,$b),
           $d := substring-after(string($i/*:link/@*:href),
                 "unique-id='"),
           $e :=substring-before($d,"'")
           for $obj in document($c)//*[@unique-id=$e] ← ❺
           return <return><intitule>{$i/*:code/text()},
             ➥{$obj/nom/text()}</intitule><aerodromeid>
             ➥{string($unique-id)}</aerodromeid>
             ➥<siteid>{$e}</siteid></return> ← ❻
```

Cette requête va chercher tous les aérodromes déclarés dans la base de données (repère ❸). Pour chacun, elle décortique le lien qui s'y trouve (série des `let` du repère ❹) et va chercher les morceaux d'information complémentaires dans les documents XML pointés par chaque structure `aerodrome` (repère ❺). Ces informations étant collectées dans les variables `$i` et `$obj`, la requête les combine sous la forme d'une nouvelle structure XML (repère ❻) qui donne le résultat suivant :

```
<return>
  <intitule>LFRM, Le Mans, Arnage</intitule>
  <aerodromeid>i21</aerodromeid>
  <siteid>i6</siteid>
</return>
<return>
  <intitule>LFRS, Nantes, Atlantique</intitule>
  <aerodromeid>i22</aerodromeid>
  <siteid>i7</siteid>
</return>
<return>
  <intitule>LFRZ, Saint Nazaire, Montoir</intitule>
  <aerodromeid>i23</aerodromeid>
  <siteid>i8</siteid>
</return>
<return>
  <intitule>LFPN, Toussus, le Noble</intitule>
  <aerodromeid>i24</aerodromeid>
  <siteid>i9</siteid>
</return>
```

Programmes du niveau entreprise (Java)

Les programmes de cette couche reçoivent les données XML retournées par les requêtes XQuery et les rendent exploitables pour des calculs génériques de transformation avant d'être remises aux programmes de la couche métier.

Nous avons vu que c'est typiquement le cas pour les coordonnées `(x,y)` des objets.

Voici le programme Java qui exécute la requête XQuery étudiée à la section précédente :

```
public Iterator searchXFromId(String id) {
   String theQuery = "for $i in document('" + Db + "')/*:aerodromes/
*:aerodrome[@unique-id='" + id + "'] \n" +
   "let $a := '" + Db + "', \n" +
   "      $b := substring-before(string($i/*:link/@*:href),'/'), \n" +
   "      $c := concat($a,$b), \n" +
   "      $d := substring-after(string($i/*:link/@*:href),
                \"unique-id='\"), \n" +
   "      $e :=substring-before($d,\"'\") \n" +
   "      for $obj in document($c)//*[@unique-id=$e] \n" +
   "      return <return>{$obj/position/x/text()}</return>";
if (dbIsConnected()) {
   Iterator result = myExecuteQuery(theQuery); ← ❼
   return result;
} else {
   return null;
}
}
```

On remarquera d'une part le changement syntaxique de la requête XQuery, lié à son utilisation dans un programme Java, et d'autre part la mise sous forme de variable (`Db`) du nom de la base XML et de l'identifiant de l'objet recherché (variable `id`). Le programme est ainsi rendu générique et peut être utilisé par plusieurs applications et bases de données différentes. L'exécution à proprement parler de la requête est confiée au programme `myExecuteQuery` (repère ❼) dont la description n'a pas d'intérêt ici.

Ce programme est accompagné des codes de transformation des coordonnées `(x,y)` en *dms* :

```
public String getDecodedXFromId(String id) {
float Xf = getFloatXFromId(id);
int Xdegre = (int) (Xf - (Xf % 1)); ← ⑨
String X = String.valueOf(Xdegre) + "°";
```

```
float Xmf = ((Xf - Xdegre) % 1) * 60 - ((((Xf-Xdegre) % 1)*60) %1); ← ⑨
int Xminute = (int) Xmf;
X = X + String.valueOf(Xminute) + "'";
int Xseconde = (int) ((Xf - Xdegre - (((float) Xminute) / 60.0))
                      * 3600); ← ⑨
X = X + String.valueOf(Xseconde) + "&quot;";
return X;
}
public float getFloatXFromId(String id) {
Iterator result = searchXFromId(id);
String X = getLitteralValueFromSimpleIterator(result); ← ❽
float Xf;
if (X != "") {
   Xf = Float.parseFloat(X); ← ❾
} else {
   Xf = 0;
}
return Xf;
}
```

Ces programmes (ici ceux pour le décodage de X – les mêmes existent pour Y) récupèrent une valeur X sous forme de chaîne de caractères à l'intérieur du DOM (Document Object Model) retourné par la requête XQuery (repère ❽), la convertissent en nombre flottant (repère ❾), puis opèrent la transformation en *dms* (repère ⑨).

Bien qu'il s'agisse d'une transformation, on se rend compte ici que la nature de cette transformation n'a aucun rapport avec ce que propose le langage XSLT : d'une part, de tels calculs numériques sont impossibles en XSLT et, d'autre part, leurs résultats doivent pouvoir être utilisés par d'autres programmes Java.

En ce qui concerne la liste des aérodromes, la requête XQuery présentée à la section précédente fait l'objet des différents programmes en action.

Voici un programme d'exécution de la requête :

```
public Iterator searchIntitulesAerodromes() {
   String theQuery = "for $i in document('" + piloteWebDb + "')
                     ➥/*:aerodromes/*:aerodrome \n" +
      "let $unique-id := $i/@unique-id, \n" +
      "     $a := '" + piloteWebDb + "', \n" +
      "     $b := substring-before(string($i/*:link/@*:href),'/'), \n" +
      "     $c := concat($a,$b), \n" +
      "     $d := substring-after(string($i/*:link/@*:href),
                  ➥\"unique-id='\"), \n" +
      "     $e :=substring-before($d,\"'\") \n" +
```

```
      "     for $x in document($c)//*[@unique-id=$e] \n" +
      "     return <return>
               <intitule>{$i/*:code/text()},{$x/nom/text()}</intitule>
               <aerodromeid>{string($unique-id)}</aerodromeid>
               <siteid>{$e}</siteid>
         </return>";
   if (dbIsConnected()) {
      Iterator result = myExecuteQuery(theQuery);
      return result;
   } else {
      return null;
   }
}
```

Un programme de décorticage des données retournées (repère ❿) sous forme de DOM :

```
public boolean créerListeDesIntitulesAerodromes() {
   Vector aTemp = new Vector();
   Vector bTemp = new Vector();
   Vector cTemp = new Vector();
   Iterator result = searchIntitulesAerodromes(); ← ❿
   if (result != null) {
      while (result.hasNext()) {
         XQueryValueIf value = (XQueryValueIf) result.next();
         NodeList subtree = value.asNode().getChildNodes();
         aTemp.addElement(subtree.item(0).getChildNodes().item(0).getNo
deValue());
         bTemp.addElement(subtree.item(1).getChildNodes().item(0).getNo
deValue());
         cTemp.addElement(subtree.item(2).getChildNodes().item(0).getNo
deValue());
      }
   }
   listeDesAerodromes = new String[aTemp.size()]; ← ⓫
   listeDesIDAerodromes = new String[bTemp.size()]; ← ⓬
   listeDesIDSitesAerodromes = new String[cTemp.size()]; ← ⓭
   aTemp.copyInto(listeDesAerodromes);
   bTemp.copyInto(listeDesIDAerodromes);
   cTemp.copyInto(listeDesIDSitesAerodromes);
   return true;
}
```

On obtient trois variables (repères ⓫ ⓬ ⓭) : les listes des noms et des identifiants primaires et secondaires des aérodromes.

On voit que ces programmes réalisent des traitements généraux indépendants de toute application spécifique (dite métier).

Programmes du niveau métier (JSP)

Ces programmes utilisent les variables calculées dans la couche précédente pour les associer et les assembler entre elles, et leur donner ainsi une logique applicative, que l'on appelle métier.

Par exemple, pour l'affichage des positions géographiques des objets, la forme JSP utilisée est la suivante :

```
<aerodrome code="<%= piloteWeb.getCodeAerodromeFromId(idAerodrome) %>"
      nom="<%= nomAerodrome %>"
      photo="<%= piloteWeb.getPhotoFromSiteId(idAerodrome) %>"
      x="<%= piloteWeb.getDecodedXFromId(idAerodrome) %>"
      y="<%= piloteWeb.getDecodedYFromId(idAerodrome) %>">
```

Une fois exécutée par le serveur Web, la page JSP produit un élément XML, tel que :

```
<aerodrome code="LFRM" nom="Le Mans, Arnage" photo="lemans.jpg"
      x="47°56'54&quot;" y="0°12'5&quot;">
```

Les valeurs x et y sont récupérées du programme Java de la couche précédente grâce aux fonctions `getDecodedXFromId(idAerodrome)` et `getDecodedYFromId(idAerodrome)` appliquées à la *bean* Java reconnue des pages JSP sous le nom `piloteWeb`.

C'est ainsi que les données sont transmises de la couche entreprise à la couche métier.

Quant à la liste des aérodromes requise pour la fabrication de la liste déroulante illustrée à la figure 7-13, voici sa programmation en JSP :

```
<% if ((piloteWeb.getAerodromes())==null)
{piloteWeb.créerListeDesIntitulesAerodromes();} %>
<% String[] nomsAerodromes = piloteWeb.getAerodromes(); %>
<% String[] nomsIdAerodromes = piloteWeb.getIdAerodromes(); %> ← ⓮
<formulaire method="get" action="planDeVolDetail.jsp">
<liste titre="Aérodrome de départ" nom="etape1">
<% for (int i=0; i<nomsAerodromes.length; i++) { %>
<option value="<%= nomsIdAerodromes[i] %>"><%= nomsAerodromes[i] %>
</option>
<% } %>
</liste>
       … … … …
```

```
<choix titre="Ajouter les points de repères proches du plan de vol"
       nom="publiePointsDeRepere"
       init="<%= piloteWeb.getPubliePointsDeRepere() %>"/>
  <choix titre="Afficher les pilotes proches de la zone"
         nom="publiePilotesProche"
         init="<%= piloteWeb.getpubliePilotesProche() %>"/>
  </formulaire>
```

Dans cet extrait, on voit la récupération de la variable indicée `nomsIdAerodromes`, calculée par le programme `getIdAerodromes` (repère ⓮). Cette liste sert ensuite à créer le menu déroulant grâce à une boucle écrite en JSP intégrée aux éléments XML de définition d'un formulaire (la boucle en question est en gras dans l'exemple).

Les balises utilisées dans cette couche métier ne sont pas encore des éléments HTML ; ces derniers ne seront générés que par la dernière couche de programmation : la couche présentation programmée en XSLT.

Figure 7–13
Le menu déroulant de PiloteWeb affichant la liste des aérodromes

Comme on peut l'observer au travers des extraits de pages JSP présentées dans cette section, les balises XML de la couche métier ne sont pas des éléments HTML. Les éléments utilisés ne sont pas identiques aux balises qui servent au stockage (nous avons pu voir comment la couche de mise en correspondance construit à la volée de nouvelles structures XML) et ne sont pas encore des balises HTML du niveau présentation. En effet, le rôle principal de la couche métier est d'assembler différentes données pour leur donner un sens du point de vue d'une application, c'est-à-dire métier.

Dans la prochaine section, nous allons étudier les transformations qui permettent de passer de la couche métier à la couche présentation.

Programmation de la présentation (XSLT)

Cette couche a pour objet d'amener les données XML à former une page HTML pleine et entière. Pour ce faire, les constructions de type métier préparées dans la couche précédente vont être transformées en HTML grâce à une programmation XSLT.

Comme nous allons le voir, la programmation XSLT peut :

- ajouter des éléments de présentation (bandeau de haut et pied de page, logos, copyright, etc.) ;
- collecter et transformer les données reçues de manière importante (réorganisation, changement de valeurs, transposition texte/image, ajout de textes standards, etc.) ;
- introduire des éléments de programmation JavaScript ;
- faire appel à une feuille de styles de type CSS activée quand la page HTML sera finalement affichée.

L'ajout d'éléments de présentation dans les pages de PiloteWeb, tels que l'avion et le nom de l'application, est typique car elle n'aurait pas lieu d'être dans les couches amont de notre programmation.

Voici la programmation XSLT correspondante :

```
<body>
<table border="0" width="600" cellspacing="0" cellpadding="0">
  <tr><td><img border="0" src="Img/piloteweb00.png"/></td>
      <td><img border="0" src="Img/piloteweb10.png"/></td>
  </tr>
  <tr>
    <td>
     <xsl:apply-templates select="login" />
    </td>
```

```
    <td class="shortcuts" valign="middle"
        border="0" background="Img/piloteweb11.png">
    <xsl:apply-templates select="raccourcis/*" />
   </td>
  </tr>
</table>
… … … …
</body>
```

On y repère l'appel des images du bandeau du haut, des boutons de connexion (`login`) ou sortie (`quitter`) de l'application et des menus (accueil, aérodrome, plan de vol). La figure 7-14 en présente le résultat.

Figure 7–14 Présentation obtenue par la transformation XSLT « fixe »

Les transformations importantes de données sont visibles sur les données météorologiques de PiloteWeb.

Figure 7–15 Affichage des données météorologiques de PiloteWeb

Le programme de transformation XSLT effectue les transformations suivantes :

- Détermination des icônes température, sens du vent et état du ciel à partir des valeurs numériques reçues. Placement de ces icônes en tête de la présentation de la météorologie.
- Concaténation des mots « Mayenne » et « France », ajout de la ponctuation.

Figure 7–16
Données sources utilisées pour les informations météorologiques

```
<meteo>
<departement>Mayenne</departement>
<pays>France</pays>
<vent unit='degrees'>WWW 270</vent>
<vitesse unit='mph'>3</vitesse>
<vitesse unit='kt'>4</vitesse>
<ciel>1cloud_fog</ciel>
<temperature unit='F'>26.1</temperature>
<temperature unit='C'>-3.3</temperature>
<humidite>65%</humidite>
<pression unit='in.Hg'>30.17</pression>
<pression unit='hPa'>1021</pression>
<visibilite unit='miles'>9</visibilite>
</meteo>
```

- Transformation de la chaîne de caractères « `WWW 270` » en « `270°` ». Le « `WWW` » sert à choisir l'icône indiquant le sens du vent.
- Changement de l'unité `mph` en `MpH`.
- Conversion des miles en kilomètres.

Le XSLT permet d'introduire des scripts JavaScript. Dans PiloteWeb, on utilise une fonction écrite en JavaScript qui s'intitule `vacopen` :

```
<script>
function vacopen(url) {
  open(url,'vac','toolbar=no,
                  location=no,
                  directories=no,
                  status=no,
                  menubar=no,
                  scrollbars=yes,
                  resizable=yes,
                  width=660,
               height=600');
  return ;
}
</script>
```

Cette fonction construit l'URL d'une ressource. Elle est utilisée dans la programmation XSLT comme suit :

```
xsl:when test="boolean(@photo)">
  <xsl:element name="a">
   <xsl:attribute name="href">
    javascript:vacopen('Img/<xsl:value-of select="@photo"/>')
   </xsl:attribute>
   <xsl:element name="img">
    <xsl:attribute name="width">300</xsl:attribute>
    <xsl:attribute name="src">Img/<xsl:value-of select="@photo"/>
    </xsl:attribute>
   </xsl:element>
</xsl:element>
</xsl:when>
```

Enfin, la programmation XSLT introduit dans le fichier HTML final l'appel à une (ou plusieurs) feuille de styles CSS. Celle utilisée dans PiloteWeb fait l'objet de la section suivante.

Programmation de l'affichage (HTML et CSS)

Le fichier HTML résultant de la transformation XSLT commence ainsi :

```
<html>
<head>
  <title>PilotWeb : le site des pilotes privés</title>
  <link rel="stylesheet" type="text/css" href="style/style.css"></link>
</head>
```

La feuille de styles `style.css` ne contient pas grand-chose, sinon quelques définitions typographiques de paragraphes. On se situe là au stade ultime des transformations subies par les données XML stockées pour qu'elles puissent être présentées à des utilisateurs finaux :

```
a {color: #b50a0a;}
p {text-align: justify;}
p.center {text-align: center;}
html, body {margin: 0px;
            padding: 0px;
            background: #b6e4fb;
            color: #2640a8;
            text-align: justify;
            font-family: helvetica,arial;
            font-size: 16px;}
```

```
.question {text-align: right;}
.reponse {font-family: helvetica, arial;
          font-size: 16px;
          color: #2640a8;}
img {border: 0px;}
.shortcuts {text-align: right;}
.titre {font-size: 18px;}
td img {display: block;}
p.meteo {font-size: 13px;
         text-align: left;}
```

En résumé…

Dans ce chapitre, nous avons montré de façon concrète deux points fondamentaux de la conception des applications :

1. La différence importante qui existe entre modèles de stockage et modèles d'affichage (ou restitution). Dans le détail de ce chapitre, nous avons même vu qu'il existait des modèles intermédiaires, volatils, encore très différents des deux autres.
2. La nécessaire organisation des couches de programmation d'une application. Certains traitements ne peuvent être réalisés qu'avec certains langages de programmation, d'autres doivent répondre à des critères de mutualisation, d'autres encore à des logiques de programmation (la couche présentation en est un exemple typique : elle doit forcément arriver à la fin des traitements).

Il est certainement utopique de penser maîtriser assez le code d'une application pour respecter totalement ces deux points, mais il est capital de tenir compte de leur existence qui tient principalement aux spécificités de XML. Cela améliorera de manière significative la possibilité de maintenance, l'extensibilité et la flexibilité de vos applications.

À partir du chapitre suivant, nous allons passer en revue des techniques de balisage que nous recommandons pour que, justement, vos applications soient le plus standard possible et le plus facile à maintenir.

Deuxième partie

Modèles de référence

8

Modèles modulaires

Nous allons maintenant décrire les techniques d'écriture particulières des modèles modulaires : l'une adaptée au cas des DTD, l'autre aux schémas XML.

Les modèles modulaires, qu'il s'agisse de DTD ou de schémas XML, poursuivent le même objectif : produire facilement plusieurs DTD ou schémas à partir d'un jeu réduit de structures. Ainsi, on obtient une flexibilité intrinsèque au modèle qui réside dans le fait qu'une modification appliquée à l'une des briques élémentaires est répercutée sur l'ensemble des schémas l'utilisant.

L'intérêt d'un tel procédé est évident mais sa mise en œuvre nécessite quelques explications.

Le cas des DTD est distinct de celui des schémas XML : leurs mécanismes de modularisation sont différents. Aussi pour les illustrer avons-nous pris des exemples représentatifs de chaque cas. Les deux exemples choisis sont célèbres. Il s'agit d'une part de la DTD DocBook destinée à l'édition de livres et d'autre part des schémas XML de la norme S1000D qui s'applique à la documentation des systèmes d'armes et de défense. Nous allons montrer deux façons de rendre modulables ces modèles :

- par le choix des éléments racines ;
- par la composition d'éléments.

La technique de composition d'éléments peut être mise en œuvre de manière mécanique avec les DTD, tandis qu'elle est forcément logique dans le cas des schémas XML. Une composition est logique quand les petits morceaux résultant du découpage sont eux-mêmes des schémas valides ; elle est mécanique dans le cas contraire.

Représentation d'un modèle modulaire

Quand un modèle est décomposé en fragments se pose la question de sa représentation : en effet, la décomposition peut ne pas correspondre à la logique du modèle conceptuel, et les représentations graphiques classiques des DTD et schémas partent toujours du principe que le modèle est « expansé ». Ces représentations ne disposent donc d'aucune fonction de mise en évidence de la modularité.

Le travail de représentation a été réalisé pour au moins l'un des deux modèles qui sert de trame à ce chapitre : la S1000D. Cette norme définit quinze classes de documents de type papier et catalogues allant des descriptions générales, le modèle `descript`, à des catalogues de pièces détachées, le modèle `IPC`.

L'approche modulaire est naturelle pour documenter des matériels :

- Une arborescence de modules peut facilement être calquée sur la structure physique ou fonctionnelle d'un matériel.
- La structure interne d'un module est générique, ce qui le rend utilisable tant pour la documentation d'un avion que pour celle d'un chargeur de batterie.

Afin d'organiser les modèles, trois niveaux de structuration ont été définis. Ils sont présentés à la figure 8-1.

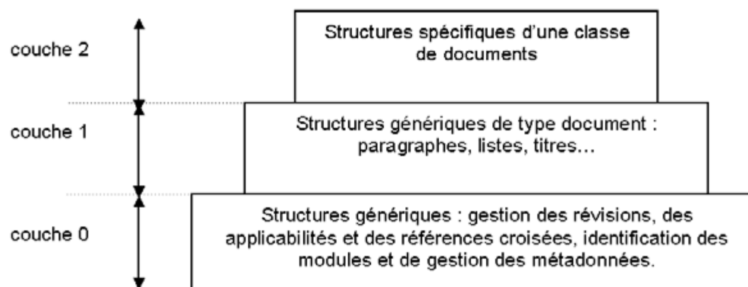


Figure 8–1
Les trois niveaux de structuration définis pour la S1000D

Le modèle est conçu pour être extensible. Cette volonté s'explique par le fait que la documentation de maintenance d'un avion ne se limite pas à l'objet physique avion, mais s'étend à tous les systèmes terrestres et embarqués qui l'accompagnent (pensons par exemple aux avions embarqués à bord de porte-avions et aux systèmes informatiques nécessaires sur le navire pour faire le relais avec ceux montés sur l'avion).

Pour éviter les redondances et recoupements, le rôle de chaque modèle de base a été parfaitement défini. Le tableau 8-1 donne un aperçu de la manière dont les modèles ont été découpés. Ce tableau doit être lu en pensant que les modèles de base ne forment pas une organisation plane mais une pyramide de modèles de base, les uns s'emboîtant dans les autres.

Tableau 8–1 Organisation des modèles de base d'un modèle modulaire
(extrait de la liste des DTD de base de la norme S1000D)

`%acrwcfg;`	`aircrew.cfg`	Configuration des composants spécifiques au type `aircrew` (équipage).
`%aecma;`	`aecma.cfg;`	Configuration des composants de base de type `aecma`.
`%af;`	`af.ent`	Décomposition des contenus de type `af` (*air vehicle fault isolation*).
`%aircrew;`	`aircrew.ent`	Décomposition des contenus de type `aircrew` (équipage).
`%base;`	`base.ent`	Composant de base des DTD S1000D.
`%capgrp;`	`capgrp.ent`	Composant de type *caption group*.
`%common;`	`common.ent`	Composants génériques.
`%comps;`	`comps.ent`	Catalogue des identifiants des différents composants.
`%descrpt;`	`descript.ent`	Décomposition des contenus de type descriptif.
`%extappl;`	`applic.ent`	Décomposition de l'applicabilité.
`%extcont;`	`content.ent`	Décomposition de base de l'élément `content`.
`%extdmc;`	`dmc.ent`	Décomposition de la codification des modules.
`%extftab;`	`fig_tab.ent`	Décomposition des éléments `figure` et `table`.
`%extlist;`	`lists.ent`	Décomposition des listes.
`%extpara;`	`paras.ent`	Décomposition des paragraphes.
`%extspec;`	`specpara.ent`	Décomposition des paragraphes spéciaux.
`%extstat;`	`status.ent`	Décomposition de l'élément `status`.
`%extstep;`	`steps.ent`	Décomposition de l'élément `steps`.
`%ipd;`	`ipd.ent`	Décomposition des contenus de type *ipd* (*illustrated parts data*).
`%nic;`	`nic.ent`	Décomposition des contenus de type *nic* (*nomenclature*, *identification*, *CSN*).
`%pmdata;`	`pmdata.ent`	Décomposition des contenus de type *pmdata* (*product management data*).
`%prelreq;`	`prelreq.ent`	Décomposition des contenus de type *prelreq* (*preliminary requirements*).
`%proced;`	`proced.ent`	Décomposition des contenus de type *procédure*.
`%schedul;`	`schedul.ent`	Décomposition des contenus de type *planification* (*schedule*).
`%wcnp;`	`wcnp.ent`	Composants des contenus de type *wcnp* (*warning*, *caution*, *note*, *paragraphs*).

Chacun de ces fragments de DTD est associé à un identifiant public et à un nom d'entité, comme on peut le voir dans la colonne de gauche du tableau (pour les schémas XML, ces identifiants publics deviendront des noms d'espaces de noms).

Une classe de documents est le résultat de l'assemblage des entités `base` de la couche 0, `extcont` de la couche 1, et enfin de celles de la couche 2 spécifiques à la classe de document considérée (par exemple, *`ipd`* pour un IPC). Les éléments du niveau 2 viennent parfois redéfinir des éléments du niveau 1 ; par exemple, la version destinée au pilote contient des listes de contrôle (plus connues sous le nom de *check-list*) qui n'apparaissent pas dans le modèle de base.

Ainsi, on peut représenter le modèle du modèle de la S1000D bien qu'il ne s'agisse pas d'un modèle conceptuel. Les figures 8-2 et 8-3 présentent les deux « macro-structures » qui font office de méta-modèles de la S1000D : celles des modèles descriptifs (figure 8-2) et des procédures (figure 8-3).

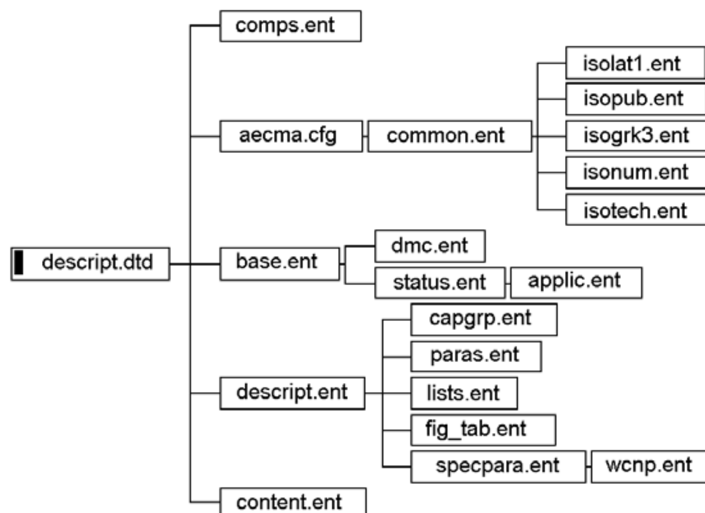


Figure 8–2
Macro-structure représentant le modèle du modèle de description de la S1000D

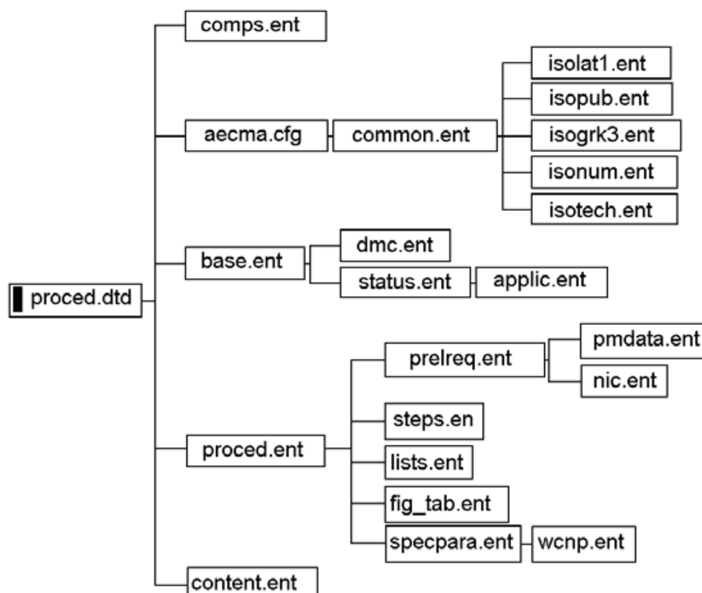


Figure 8–3
Macro-structure représentant le modèle du modèle de procédures de la S1000D

Cette structuration des modèles a également été suivie pour la définition des feuilles de styles XSL associées aux modules. En effet, la modularité des modèles doit s'accompagner d'une modularité équivalente des programmes associés.

Cas des DTD

Dans ce chapitre, nous présentons les techniques de modularisation des DTD. Nous examinerons tout d'abord un mécanisme basé sur le choix des éléments racines, puis un autre établi sur la composition des éléments.

Choix des éléments racines

En SGML et XML 1.0, on peut sans problème écrire des DTD permettant aux documents XML instances d'avoir comme élément racine tout élément du modèle. Nous allons rappeler comment fonctionne ce mécanisme de base.

Soit une DTD simple :

```
<!ELEMENT sites (site*)>
<!ELEMENT site (nom, photo?, position)>
<!ATTLIST site unique-id ID #REQUIRED>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT photo EMPTY>
<!ATTLIST photo nom CDATA #REQUIRED
          format (gif|jpg|png) "gif">
<!ELEMENT position (x,y)>
<!ELEMENT x (#PCDATA)>
<!ELEMENT y (#PCDATA)>
```

Cette DTD définit sept éléments : `sites`, `site`, `nom`, `photo`, `position`, `x`, `y`.

Dans les documents XML conformes à cette DTD, on déclare l'élément racine en utilisant la carte DOCTYPE en début de document, comme ci-après.

Premier exemple, l'élément racine est `sites` :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sites SYSTEM "sites.dtd" []> ← sites est déclaré ici comme
                                         élément racine
<sites>
<site unique-id="i1">
<nom>Pilote de Démo</nom>
<photo nom="pingouin.jpg" format="jpg"/>
```

```
<position>
  <x>48.7494</x>
  <y>-2.1108</y>
</position>
</site>
</sites>
```

Une fois la déclaration faite, l'instance DOIT commencer par la balise `<dmodule>`.

Dans notre second exemple, l'élément terminal `code` est la racine. On obtient une instance minimale, mais tout autant valide que celle du premier exemple par rapport à notre DTD.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE code SYSTEM "exempleDTD.dtd" []> ← l'élément racine ici
                                             déclaré est code
<code>Text</code>
```

VOCABULAIRE Élément terminal

En XML, un élément terminal est un élément qui ne contient pas lui-même d'autres éléments. Son seul contenu peut être du texte ou rien du tout (cas des éléments vides).

VOCABULAIRE Feuille

Une structure XML étant assimilée à un arbre, les éléments terminaux sont également appelés les feuilles de cet arbre.

APARTÉ Une différence entre SGML et XML

Puisque l'on parle de DTD, il faut noter qu'en XML, la carte DOCTYPE ne peut être utilisée que dans les instances. Elle est interdite dans les DTD. Ce détail change singulièrement les possibilités d'attribution de l'élément racine : en SGML, la carte DOCTYPE pouvait être utilisée dans les DTD, autorisant ainsi les concepteurs à imposer aux utilisateurs l'élément racine à employer. Cependant, chaque DTD ne pouvait se voir attribuer qu'un seul élément racine.

Avec ce mécanisme, les concepteurs de DTD ne peuvent pas imposer aux utilisateurs de démarrer leurs documents XML par un élément précis : tous sont permis. Rien n'est donc contrôlé. C'est une forme de flexibilité qui ne correspond pas au souhait légitime des concepteurs de DTD en matière de contrôle de la modularité de leurs modèles.

Composition d'éléments

Le modèle de la DTD DocBook a cette caractéristique remarquable qu'un très grand nombre d'éléments et d'attributs y sont définis.

> **HISTOIRE Le modèle DocBook**
>
> Le modèle DocBook date de 1991 et a traversé les époques SGML, HTML, XML et enfin XML Schema. Son origine est à la fois technique et éditoriale puisque ses concepteurs étaient respectivement le fabricant d'ordinateurs Hal Computer Systems et la maison d'édition O'Reilly & Associates. Ils se sont inspirés des travaux conduits à l'époque par l'OSF (Open Software Foundation) pour l'interopérabilité d'Unix : SGML était alors envisagé comme format d'écriture des manuels de référence des versions OSF d'Unix.
>
> Le modèle initial a été rapidement repris par le groupe de Davenport, du nom du lieu où se tint sa première réunion, réunissant les producteurs de documents et livres sur l'informatique : O'Reilly, Novell et Digital. Le modèle DocBook devint officiellement l'affaire du groupe de Davenport en 1994.
>
> La dernière version officielle de DocBook est la V4.3, en date du 31 mars 2004. Celle que nous utilisons dans ce chapitre est la V4.2 du 17 juillet 2002. Le modèle DocBook est l'un des rares modèles à être disponible dans tous les langages de modélisation possibles : SGML et XML 1.0 pour les versions officielles, XML Schema, RelaxNG et Trex pour les versions non officielles. Ainsi, on peut donc comparer ces différents langages de modélisation.

Avec DocBook, on entre dans le domaine des méga-DTD. Les statistiques concernant ce modèle sont impressionnantes : 388 éléments y sont définis, dont 185 mixtes, 18 vides et 21 récursifs. Sur ces éléments, pas moins de 4548 attributs sont déclarés, dont 109 différents (en nom et en type). La version XML Schema fait 8249 lignes…

Ce qui est intéressant, c'est la manière dont le modèle gère cette complexité. Sans aller jusqu'à parler de la flexibilité des documents XML écrits en DocBook, la question posée ici est véritablement celle de la flexibilité du modèle lui-même !

La technique de gestion utilisée est différente selon le langage de modélisation utilisé. Pour les versions SGML et XML, la modularité du modèle est gérée avec la technique des sections marquées : la totalité du modèle tient dans un seul fichier, dont certaines parties peuvent être incluses ou exclues grâce au mécanisme des sections marquées de SGML et XML.

La XML Schema ne peut utiliser ce mécanisme qui n'existe pas dans cette version : elle s'appuie sur des fichiers contenant des sous-schémas. Nous aborderons cette question à la prochaine section de ce chapitre.

Nous présentons ci-après quelques traits caractéristiques de la technique des sections marquées. Nous commençons avec un extrait de la DTD bi-format SGML/XML : une DTD qui contient les syntaxes SGML et XML d'écriture de DTD.

Dans le premier extrait exposé ci-après, l'initialisation principale gouverne la totalité du jeu d'instructions permettant de passer d'un langage à l'autre. Les deux entités paramètres `sgml.features` et `xml.features` sont initialisées avec les valeurs `IGNORE` et `INCLUDE`. Ces entités paramètres s'excluent mutuellement (si l'une vaut `IGNORE`, l'autre doit valoir `INCLUDE`, et vice-versa). Cela permet de basculer de la version XML (quand `xml.features` vaut `INCLUDE`) à la version SGML de la DTD (quand `sgml.features` vaut `INCLUDE`).

Dans l'extrait ci-après, remarquez comment les deux entités paramètres sont immédiatement mises en œuvre comme critères de sélection des sections marquées (repère ⓿).

```
<!ENTITY % sgml.features "IGNORE">
<![%sgml.features;[ ← ⓿
<!ENTITY % xml.features "IGNORE">
]]>
<!ENTITY % xml.features "INCLUDE"> ← ❶

<![%sgml.features;[ ← ❷
<!ENTITY % ho "- O">
<!ENTITY % hh "- -">
]]>
<![%xml.features;[ ← ❸
<!ENTITY % ho "">
<!ENTITY % hh "">
]]>
```

Explications :

L'entité paramètre `sgml.features` est initialisée :

- Si sa valeur est `IGNORE`, la section marquée repérée ⓿ est ignorée du parseur qui passe directement à l'initialisation de l'entité paramètre `xml.features` qui vaudra alors `INCLUDE`.
- Si sa valeur est `INCLUDE`, la section marquée repérée ⓿ est prise en compte par le parseur qui initialise l'entité paramètre `xml.features` avec la valeur `IGNORE`. La seconde initialisation de cette dernière entité paramètre (repère ❶) ne sera pas prise en compte par le parseur car une règle établit que c'est toujours la première initialisation d'une entité paramètre qui fait sa valeur finale.

> **VOCABULAIRE Entité paramètre**
>
> Une entité paramètre est une variable qui n'a le droit d'être utilisée que dans une DTD. On la reconnaît grâce au caractère % qui suit immédiatement le mot-clé `ENTITY`.

Les repères ❷ et ❸ correspondent à des sections marquées, montrant comment les entités paramètres définies au début peuvent servir à définir de manière conditionnelle d'autres entités paramètres, qui, à leur tour, correspondent à des morceaux de DTD.

En l'occurrence, on montre comment une différence de syntaxe entre DTD SGML et DTD XML peut être prise en compte. Le repère ❷ correspond à la syntaxe SGML : utilisation dans les DTD de la chaîne de caractères "- O" pour indiquer les minimisations de balisage autorisées. Le repère ❸ correspond à celle de XML où, ces minimisations n'étant pas autorisées, la chaîne de caractères correspondante est tout simplement vide.

Ce mécanisme, qui rend possible la manipulation physique de la DTD (dans notre exemple, nous supprimons ou ajoutons physiquement des chaînes de caractères dans la DTD), peut être utilisé pour la modifier logiquement : on peut ainsi rendre variable le modèle que la DTD définit.

Nous allons expliquer la façon dont ce mécanisme a été utilisé dans la DocBook après en avoir donné un exemple caractéristique.

Voici un extrait de la DTD qui va nous servir d'exemple :

```
<!ENTITY % local.docinfo.char.class ""> ← ❺
<!ENTITY % docinfo.char.class "author|authorinitials|corpauthor
                               |modespec|othercredit|productname|
                               productnumber|revhistory
                               %local.docinfo.char.class;">
```

Cet extrait est typique de la manière dont la DTD DocBook a été écrite : une *classe d'éléments* (ici, la classe `docinfo.char.class`) est une entité paramètre qui représente un fragment de modèle de contenu.

Ce dernier est extensible de la manière suivante :

- Une entité « compagnon » est adjointe à l'entité principale : `local.docinfo.char.class`.
- Dans la DTD de base, cette entité est vide (repère ❺).
- Il suffit de la faire précéder d'une autre déclaration pour que sa nouvelle valeur soit prise en compte. En effet, comme c'était déjà le cas avec les DTD SGML, c'est la première initialisation d'une entité qui fige le contenu des DTD de XML 1.0.

Ainsi, avec la série de déclarations suivantes, le contenu de l'entité `docinfo.char.class` sera "`author | authorinitials | corpauthor | modespec | othercredit | productname | productnumber | revhistory | isbn | issuedate | pubdate`" :

```
<!ENTITY % local.docinfo.char.class "|isbn|issuedate|pubdate">
```

```
<!ENTITY % local.docinfo.char.class "">
<!ENTITY % docinfo.char.class "author|authorinitials|corpauthor
                               |modespec|othercredit|productname|
                               productnumber|revhistory
                               %local.docinfo.char.class;">
```

En XML Schema, cette technique est impossible. Le concept d'entité paramètre n'existe tout simplement pas et, pour des raisons qui sont faciles à expliquer, aucun mécanisme n'a été prévu pour compenser cette absence.

INFORMATION Pourquoi le mécanisme des entités paramètres n'existe-t-il pas en XML Schema ?

XML Schema repose sur des constructions parfaitement logiques de modèles. Les sous-schémas doivent être eux-mêmes des schémas valides, indépendamment de tout autre schéma. Le mécanisme des entités paramètres qui fait qu'on peut mettre n'importe quel bout physique de DTD dans n'importe quelle variable ne correspond pas du tout à cette logique. C'est pourquoi le mécanisme des entités paramètres n'a pas de raison d'être en XML Schema.

C'est pourquoi la logique mise en œuvre pour construire les DTD de DocBook n'a pas pu être transposée, le mécanisme des entités paramètres faisant défaut. Dans la version XML Schema de DocBook, le modèle `docinfo.char.class` n'est pas extensible :

```
<xsd:group name="docinfo.char.class">
  <xsd:choice>
   <xsd:element ref="db:author"/>
   <xsd:element ref="db:authorinitials"/>
   <xsd:element ref="db:corpauthor"/>
   <xsd:element ref="db:modespec"/>
   <xsd:element ref="db:othercredit"/>
   <xsd:element ref="db:productname"/>
   <xsd:element ref="db:productnumber"/>
   <xsd:element ref="db:revhistory"/>
 </xsd:choice>
 </xsd:group>
```

Afin de gérer la complexité des imbrications possibles d'entités paramètres, les cas de figure gérés par DocBook sont savamment décrits dans des tableaux de configuration tels que celui que nous fournissons ci-après. Ils ne représentent pas directement les modèles de contenus finaux mais plutôt des macro-structures, qui pourraient porter le nom de formes architecturales de la DTD, elles-mêmes entités paramètres utilisées dans d'autres macro-structures. Un exemple vient plus bas à l'appui de notre propos.

Tout d'abord, commençons par montrer le tableau qui sert à exprimer les combinaisons autorisées de macro-structures :

```
<!--
                  list admn line synp para infm form cmpd gen desc
Component          X    X    X    X    X    X    X    X    X    X
Sidebar            X    X    X    X    X    X    X    a    X
Footnote           X         X    X    X    X
Example            X         X    X    X    X
Highlights         X    X              X
Paragraph          X         X    X         X
Admonition         X         X    X    X    X    X    b    c
Figure                       X    X         X
Table entry        X    X    X         X    d
Glossary def       X         X    X    X    X         e
Legal notice       X    X    X         X    f
-->
```

Puis, continuons en donnant un exemple d'une de ces macro-structures :

```
<!ENTITY % local.component.mix ""> ← Entité utilisée dans une
                                      macrostructure
<!ENTITY % component.mix ← Macrostructure ou forme architecturale
                            de modélisation
 " %list.class; | %admon.class; | %linespecific.class; | %synop.class;
 | %para.class; | %informal.class; | %formal.class; | %compound.class;
 | %genobj.class; | %descobj.class; | %ndxterm.class; | beginpage
  %local.component.mix;"> ← Utilisation de l'entité
                             local.component.mix
```

Au final, les macro-structures sont utilisées dans des définitions d'éléments dans lesquelles elles sont combinées avec des déclarations d'éléments écrites en dur. Dans l'exemple ci-après, la macro-structure `component.mix` est combinée à l'élément `title` :

```
<!ELEMENT msgexplan %ho; (title?, (%component.mix;)+)>
```

> **ASTUCE Truc technique**
>
> Dans la DTD DocBook, la compatibilité SGML/XML est assurée, entre autres, par l'utilisation de l'entité ho. Cette dernière rend variables les options de minimisation qui existent en SGML et pas en XML. Le mécanisme régissant cette entité est présenté au début de la section.

En version XML Schema, le modèle de contenu de cet élément est défini ainsi :

```
<xsd:complexType name="msgexplan">
  <xsd:sequence>
    <xsd:element ref="db:title" minOccurs="0"/>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:group ref="db:component.mix"/> ← ❶
    </xsd:choice>
  </xsd:sequence>
  <xsd:attributeGroup ref="db:common.attrib"/> ← ❶
  <xsd:attributeGroup ref="db:msgexplan.role.attrib"/> ← ❶
</xsd:complexType>
```

On y trouve la présence de l'élément `title` et du groupe `component.mix`. Remarquez que les objets composant ce type, le groupe comme les attributs, sont définis globalement dans un espace de noms spécifique (identifié par le préfixe `db`). Dans la définition du type complexe `msgexplan`, ils ne sont que référencés (repère ❶).

Le mécanisme est identique pour les attributs.

Cas des schémas XML

Choix des éléments racines

Contrairement à XML 1.0, XML Schema prévoit des mécanismes pour limiter le choix des éléments racines. Cela donne trois cas de figure :

- Un seul élément est racine.
- Tous les éléments peuvent être racines.
- *N* éléments des *m* définis par le schéma peuvent être racines.

Il suffit pour cela d'utiliser le mécanisme des déclarations globales et locales. Avec les schémas XML de la recommandation XML Schema du W3C, tout élément défini globalement peut être racine d'un document XML valide par rapport à ce schéma, tandis qu'aucun élément défini localement ne peut l'être.

Nous allons présenter ci-après quelques exemples, après conversion de la DTD `sites`, utilisée en schéma XML.

Exemple : un schéma XML où tous les éléments sont globaux :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
          elementFormDefault="qualified">
 <xs:element name="nom" type="xs:string"/>
 <xs:element name="photo">
  <xs:complexType>
   <xs:attribute name="nom" type="xs:string" use="required"/>
   <xs:attribute name="format" default="gif">
    <xs:simpleType>
     <xs:restriction base="xs:NMTOKEN">
      <xs:enumeration value="gif"/>
      <xs:enumeration value="jpg"/>
      <xs:enumeration value="png"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
  </xs:complexType>
 </xs:element>
 <xs:element name="position">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="x" ref="x"/>
    <xs:element name="y" ref="y"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="site">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="nom" ref="nom"/>
    <xs:element name="photo" ref="photo" minOccurs="0" maxOccurs="1"/>
    <xs:element name="position" ref="position"/>
   </xs:sequence>
   <xs:attribute name="unique-id" type="xs:ID" use="required"/>
  </xs:complexType>
 </xs:element>
 <xs:element name="sites">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="site" ref="site" minOccurs="0"
        maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="x" type="xs:string"/>
 <xs:element name="y" type="xs:string"/>
</xs:schema>
```

Ce schéma permet, comme dans le cas des DTD XML, d'utiliser comme racine des documents XML tout élément défini dans le schéma. Par exemple, on peut utiliser indifféremment les éléments `sites` et `code`, et obtenir les documents valides suivants :

```
<?xml version="1.0" encoding="UTF-8"?>
<sites xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="sites.xsd"> ← la racine est sites
<site unique-id="i1">
<nom>Pilote de Démo</nom>
<photo nom="pingouin.jpg" format="jpg"/>
<position>
  <x>48.7494</x>
  <y>-2.1108</y>
</position>
</site>
</sites>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<code xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="sites.xsd"> ← la racine est code
contenu textuel de l'élément</code>
```

Composition d'éléments

Pour présenter le mécanisme de composition d'éléments avec les schémas XML, nous allons nous appuyer sur les modèles réalisés par le groupe de travail EPWG de la norme S1000D.

Un peu comme avec les DTD, la solution retenue ici consiste à ce que plusieurs schémas principaux soient définis par des combinaisons sophistiquées de schémas de base. Ainsi, si l'élément racine des documents XML conformes à la S1000D est toujours l'élément `dmodule`, on compte plus d'une dizaine de schémas XML différents (définissant autant de types de documents différents) qui résultent de la combinaison d'environ quatre-vingts schémas de base.

Le modèle repose sur les postulats suivants :

- Aucun espace de noms cible particulier n'est défini afin de conserver une totale modularité du modèle lui-même (repère ⓿).
- La modularité est assurée par des inclusions de schémas de base (repère ❶).
- Les modèles de contenu des éléments sont définis en utilisant le mécanisme des groupes, et non en suivant les mécanismes de dérivation de types de XML Schema (repère ❷). Les noms de groupes sont ainsi utilisés comme des

variables, ce qui donne un effet similaire à celui qu'on a pu obtenir avec les entités paramètres des DTD (voir section précédente).

Voici le schéma principal des documents de type « crew », réduit à l'assemblage de schémas de base au moyen du mécanisme d'inclusion de XML Schema (repère ❶).

```
<?xml version="1.0"?>
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"> ← ⓿
<xs:include schemaLocation="project.cfg"/>  ← ❶
<xs:include schemaLocation="common.xsd"/>   ← ❶
<xs:include schemaLocation="base.xsd"/>     ← ❶
<xs:include schemaLocation="capgrp.xsd"/>   ← ❶
<xs:include schemaLocation="crew.xsd"/>     ← ❶
<xs:include schemaLocation="fig_tab.xsd"/>  ← ❶
<xs:include schemaLocation="content.xsd"/>  ← ❶
<xs:include schemaLocation="paras.xsd"/>    ← ❶
</xs:schema>
```

Parmi les schémas de base, l'un se singularise, il s'agit de `crew.xsd`. En effet, c'est ce schéma qui déterminera la véritable structure finale des documents de type « crew ». Nous n'en donnons ici que les extraits les plus intéressants pour notre propos.

```
<?xml version="1.0"?>
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"> ← ⓿
<xs:include schemaLocation="heading.xsd"/> ← ❶
<xs:include schemaLocation="wcnp.xsd"/>    ← ❶
<xs:include schemaLocation="paracon.xsd"/> ← ❶
<xs:include schemaLocation="lists.xsd"/>   ← ❶
<xs:group name="CONTENT">                  ← ❷
  <xs:sequence>
   <xs:element ref="acrw"/>
  </xs:sequence>
 </xs:group>

 <xs:group name="NPAR">                    ← ❷
  <xs:sequence>
   <xs:element ref="warning" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element ref="caution" minOccurs="0" maxOccurs="unbounded"/>
   <xs:group ref="NP" minOccurs="0"/>      ← ❷
   <xs:element ref="drill" minOccurs="0"/>
  </xs:sequence>
 </xs:group>

… … …
```

```
 <xs:element name="drill">
  <xs:complexType>
   <xs:choice maxOccurs="unbounded">
    <xs:group ref="fft_inc" minOccurs="0" maxOccurs="unbounded"/> ← ❷
    <xs:group ref="DRLINTRO"/> ← ❷
    <xs:choice minOccurs="0" maxOccurs="unbounded">
     <xs:group ref="STEPS" maxOccurs="unbounded"/> ← ❷
     <xs:element ref="subdrill" minOccurs="0" maxOccurs="unbounded"/>
    </xs:choice>
    <xs:element ref="endmattr" minOccurs="0"/>
   </xs:choice>
   <xs:attribute ref="drilltyp"/>
   <xs:attribute name="ordered" default="on">
    <xs:simpleType>
     <xs:restriction base="xs:NMTOKEN">
      <xs:enumeration value="on"/>
      <xs:enumeration value="off"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
   <xs:attributeGroup ref="bodyatt"/> ← ❷
   <xs:attributeGroup ref="secur"/> ← ❷
  </xs:complexType>
 </xs:element>

… … …

</xs:schema>
```

Voici enfin le schéma de base `base.xsd` qui, contrairement à ce que l'on pourrait penser, fait appel à d'autres schémas de base (repère ❶).

```
<?xml version="1.0"?>
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<xs:import namespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   schemaLocation="rdf.xsd"/>
<xs:include schemaLocation="dmc.xsd"/>     ← ❶
<xs:include schemaLocation="pmc.xsd"/>     ← ❶
<xs:include schemaLocation="status.xsd"/>  ← ❶
<xs:element name="dmodule">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="rdf:Description" minOccurs="0"/>
    <xs:element ref="idstatus"/>
```

```
    <xs:element ref="content"/>
   </xs:sequence>
   <xs:attribute name="id" type="xs:ID"/>
  </xs:complexType>
 </xs:element>

 <xs:element name="idstatus">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="dmaddres"/>
    <xs:element ref="status"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="techname" type="xs:string"/>
 <xs:element name="infoname" type="xs:string"/>

 <xs:simpleType name="NONNEGI2">
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:totalDigits value="2" fixed="true"/>
  </xs:restriction>
 </xs:simpleType>

 <xs:element name="issdate">
 <xs:complexType>
   <xs:attributeGroup ref="DATE"/>   ← ❷
  </xs:complexType>
 </xs:element>

… … …
</xs:schema>
```

La difficulté que présente une telle organisation réside bien sûr dans la constitution de la pyramide de schémas de base ; en effet, les modèles définis à la base de la pyramide doivent être utilisés par des modèles des étages supérieurs sans risque de collision ni de boucle. Cela serait le cas si un modèle du niveau de base utilisait des constructions définies dans des modèles de niveau supérieur.

XML Schema impose un contrôle parfait des schémas de base. En effet, une règle importante de ce langage dit que chaque modèle, quels que soient sa taille et son rôle dans l'architecture des modèles, doit être valide. Tout schéma de base doit ainsi obligatoirement être un schéma valide, indépendamment de tout schéma l'utilisant.

En résumé...

La création de schémas modulaires pouvant se composer les uns avec les autres ne relève pas d'un mécanisme de dérivation de type, contrairement à ce que pourrait laisser croire l'usage commun. Pour ce faire, on utilise la technique des types/éléments paramétrables (`template class` en UML). En DTD, ce rôle est dévolu aux `ENTITY`. En XML Schema, un mécanisme équivalent est disponible au moyen des groupes.

Dans ce chapitre, nous avons étudié la manière d'écrire des DTD et schémas modulaires, soit en agissant sur les éléments racines, soit en écrivant les modèles de manière à composer les éléments.

Pour définir des modèles modulaires, on factorise ce qui se trouve être commun à plusieurs modèles. Le seul véritable risque, c'est d'avoir des difficultés à mesurer les conséquences d'une modification d'un modèle de base sur l'ensemble des autres modèles. Aussi, nous avons montré dans ce chapitre comment représenter les modèles de manière macroscopique.

Le mécanisme des entités paramètres qu'on peut utiliser pour les DTD modulaires n'a pas été reconduit en XML Schema et il faut utiliser d'autres artifices pour obtenir un résultat similaire. La solution la plus simple et la plus flexible est de passer par des groupes.

Nous verrons dans le chapitre suivant une variante de l'approche modulaire pour les modèles. Elle consiste à intégrer dans les modèles des éléments qui n'ont d'autre rôle que de découper le modèle en cas de besoin.

9

Éléments purement structurels

Ce chapitre va être l'occasion de travailler sur un concept important, celui d'éléments purement structurels. Nous allons notamment nous intéresser :

- au rôle et à l'importance des éléments purement structurels ;
- à la procédure à suivre pour identifier les éléments purement structurels d'un schéma XML.

L'expression « élément purement structurel » ne fait pas partie de la terminologie propre à XML Schema, et n'appartient d'ailleurs pas à une quelconque terminologie officielle du monde XML. Elle a été élaborée en 1993, à une époque où l'on ne parlait pas encore de XML, ni de XML Schema, et encore moins de document XML bien formé et d'espaces de noms.

HISTOIRE Élément purement structurel

Cette expression a été élaborée en 1993 par un groupe de travail sur les bases de données SGML réunissant Ludovic van Vooren et l'un des auteurs de cet ouvrage, Jean-Jacques Thomasson, qui travaillaient à l'époque pour le compte de la société Interleaf. Ludovic van Vooren est connu de la communauté SGML pour avoir développé Mark-It, premier parseur SGML, sous l'égide d'un projet commandé à la société Sema Group Belgium par la communauté européenne.

Cette terminologie a donc l'avantage d'être affranchie de toute attache directe avec une application particulière de XML, qu'il s'agisse d'un mécanisme, d'une recommandation ou d'un vocabulaire particulier. Cela lui permet d'être l'unique représentant d'une théorie et d'une méthode d'analyse qui se veulent universelles, c'est-à-dire valables pour n'importe quelle application XML, et même au-delà du seul monde XML. La théorie que nous allons étudier dans ce chapitre permet de mieux comprendre comment l'information, au sens le plus large du terme, est, ou peut être, structurée.

La théorie des éléments purement structurels offre un point de vue autorisant l'élaboration de meilleurs modèles XML.

Il s'agit d'éléments XML dont le rôle dans la structure est singulier : en effet, ils ont un pouvoir structurant supérieur aux autres. Ils sont à un schéma XML ce que nos propres articulations sont au squelette humain : ils donnent de la souplesse à la structure tout en amarrant solidement les membres entre eux.

Après avoir rappelé les origines du concept, nous montrerons la procédure qui permet, dans un schéma XML existant, de découvrir parmi les éléments ceux qui sont purement structurels.

Genèse de la théorie des éléments purement structurels

Pour présenter le concept d'élément purement structurel, nous allons partir du principe que les structures en forme de poupées russes de XML sont comparables à une arborescence de fichiers, et gager ainsi d'une hypothétique (et trompeuse…) analogie entre les deux.

Dans le tableau 9-1, nous exposons un fragment XML et, à sa droite, sa représentation sous forme de dossiers et de fichiers.

Dans la suite de ce chapitre, nous détaillerons cette représentation de la structure, et montrerons pourquoi la transposition proposée ici n'est pas réaliste et ne peut donc pas être systématisée. À ce stade, les éléments intermédiaires de la structure ont été représentés par des répertoires, les éléments terminaux par des fichiers, et les attributs n'ont pu être représentés.

En cherchant à représenter ainsi un document XML, on soulève immanquablement les questions suivantes :

1. Peut-on assimiler une balise XML à un fichier ?
2. Où doit-on mettre les attributs ?
3. Que faire des métadonnées qui s'appliquent à tout le document ?

Tableau 9–1 Analogie entre un fragment XML et un système de fichiers

Fragment XML type	Analogie avec un système de fichiers

`<doc>`
`<volume tocentry="1">`
`  <docpart>`
`   <front security="u">`
`    <idinfo verified="0" security="u"/>`
`    <revnum security="u"/>`
`   </front>`
`  </docpart>`
`</volume>`
`</doc>`
	doc
	volume
	docpart
	front
	idinfo.txt
	revnum.txt

Le premier point tient du bon sens. Il provient d'une réflexion simple qui consiste à se demander s'il est concevable de gérer un document XML en le fragmentant jusqu'au plus petit de ses éléments. En d'autres termes, faut-il vraiment attendre d'avoir atteint le niveau terminal pour passer à un stockage sous la forme de fichiers ?

Le deuxième point tient à la forme du XML : l'analogie avec un système de fichiers permet de représenter visuellement la structure d'un document XML. Or, la ramener à une série d'éléments emboîtés les uns dans les autres est réducteur. Les attributs contiennent une information qui est parfois aussi importante que celle des éléments.

Le troisième point est une généralisation du problème de la représentation des attributs du point précédent. Les métadonnées d'un document sont généralement constituées par des éléments placés en début de document XML. Leur rôle est de qualifier l'ensemble du document ou des éléments du document en particulier. Rapidement, on s'aperçoit qu'il s'agit d'éléments qui, bien que frères ou enfants directs d'autres éléments, les qualifient. Ces éléments jouent le rôle d'attributs : la logique XML de l'arborescence est rompue.

Comme vous allez le découvrir dans les sections suivantes, l'étude de ces trois points met en évidence trois catégories d'éléments : les *éléments de données*, les *éléments de données globaux* et les *éléments purement structurels*.

La réflexion que nous proposons dans les sections suivantes n'a d'autre objet que de faire comprendre pourquoi les éléments d'un document XML ne sont pas tous égaux, contrairement à ce que l'on pourrait penser à première vue.

Peut-on assimiler une balise XML à un fichier ?

Quand on cherche à gérer des modules d'information, on se pose tôt ou tard la question de savoir quelle est la taille idéale d'un module. Doit-on aller jusqu'à des granules minuscules ou faut-il qu'ils aient une certaine taille ?

Un document XML a ceci de particulier qu'il peut contenir des balises jusqu'à la plus infime des données, au moindre des caractères. Or, cela a-t-il un sens de gérer de si petites unités d'information ?

Pour répondre à cette question, nous pourrions répondre que cela dépend de la nature de la donnée, de l'application, et qu'il faut juger au cas par cas. Toutefois, cette réponse ne permet pas de mettre en place une quelconque technique objective d'analyse des schémas. La réponse doit être trouvée par la seule analyse du schéma.

Pour cela, nous allons nous souvenir que XML Schema reconnaît quatre types d'éléments : à contenu simple, à contenu complexe, à contenu mixte et enfin à contenu vide. Le tableau 9-2 rappelle ce que signifie chacun de ces cas.

Tableau 9–2 Les quatre types d'éléments de XML Schema

	Type de contenu	
Type d'élément	**Présence de texte**	**Présence de sous-éléments**
Élément simple	Oui	Non
Élément complexe	Non	Oui
Élément mixte	Oui	Oui
Élément vide	Non	Non

En tentant la correspondance balise/fichier du point de départ de notre réflexion, on s'aperçoit qu'il n'est pas de solution simple. Nous allons le constater au cas par cas dans les sections suivantes.

Cas des éléments simples

Ces éléments ne contiennent que du texte, lequel peut être vide quand le concepteur du schéma XML l'a autorisé (en ayant précisé `nillable=true` dans la définition de l'élément).

Puisqu'un texte ne peut être stocké ailleurs que dans un fichier, les éléments de type simple sont obligatoirement assimilés à des fichiers et la seule question est de savoir si la balise doit être elle-même incluse dans le fichier ou si le nom de l'élément doit devenir celui du fichier. Ces deux possibilités sont illustrées au tableau 9-3.

Rien ne permet de pencher en faveur de l'une ou de l'autre de ces deux représentations, sauf qu'on pourra noter que la seconde représentation dissocie l'élément `p` de son contenu textuel.

Les éléments de type simple sont des *éléments de données*.

Tableau 9–3 Représentation d'un élément simple par un fichier

Exemple d'élément à contenu simple	Première représentation possible
L'utilitaire CATSTA est utilisé pour traiter des données statistiques et comptables sous VM et MVS.	Un fichier portant un nom banal contient l'ensemble du fragment, c'est-à-dire le texte et les deux balises : fichier.txt
	Seconde représentation possible
	Le fichier porte le nom de l'élément (p) et ne contient que le texte du fragment : p.txt

Cas des éléments complexes

Alors que les éléments de type simple sont les extrémités, ou feuilles, de l'arbre que représente un document XML, les éléments complexes en constituent les branches et les ramifications. Un type complexe ne peut avoir comme enfant que des sous-éléments. Si, dans notre analogie, la correspondance avec un dossier semble être acquise, nous allons voir qu'il n'en est rien. L'analogie entre un élément complexe et un dossier dépend en réalité de la nature de ses sous-éléments. Trois cas de figure peuvent se présenter, lesquels sont détaillés au tableau 9-4.

Tableau 9–4 Représentations d'un élément complexe par un système de fichiers

Description	Représentation graphique
Premier cas : tous les sous-éléments ont des contenus complexes. L'élément `volume` n'est composé que de sous-éléments complexes, aucun d'eux ne contient directement du texte. Ils sont assimilables à des dossiers. Nous obtenons donc une représentation arborescente classique, constituée de dossiers et de sous-dossiers.	volume docfront docpart docrear
Deuxième cas : au moins un sous-élément a un contenu simple, les autres ayant un contenu complexe. Le sous-élément de type simple est assimilable à un fichier, les autres à des dossiers. Le dossier `volume` contient ici un fichier (il s'agit de l'icône `fichier.txt`) et des sous-dossiers qui représentent des sous-éléments de type complexe.	volume fichier.txt docpart docrear
Troisième cas : tous les sous-éléments sont de type simple et sont assimilables à des fichiers. Le dossier `volume` ne contient que des fichiers.	volume idinfo.txt revnum.txt warnpage.txt

Dans le premier cas, l'élément `volume` ne contient que des sous-dossiers ; il n'y a aucun problème pour que cet élément soit un dossier lui-même : dans notre terminologie, il s'agit d'un *élément purement structurel*.

Dans le deuxième cas, l'élément `volume` contient un mélange de dossiers et fichiers. Nous disons que c'est un *élément de données global*, locution vague pour désigner une situation où l'objet n'est ni un pur conteneur ni un vrai fichier.

Dans le troisième cas, le dossier `volume` ne contient que des fichiers. Il pourrait être un fichier lui-même ; il suffirait pour cela de concaténer les trois sous-fichiers. Nous disons dans ce cas que l'élément est un *élément de données*.

La nature d'un élément à contenu complexe dépend de ses enfants. C'est eux et eux seuls qui conditionnent la nature de l'élément, et non ses parents directs ou ses prédécesseurs.

Cas des éléments mixtes

Le contenu d'un élément mixte est un mélange de texte et sous-éléments. Les représenter graphiquement selon le modèle dossier/fichier est donc plus délicat. Ce modèle est typique du monde des documents ; il constitue une différence majeure entre les applications de XML destinées aux bases de données et celles destinées à la documentation au sens papier.

Dans le tableau 9-5, nous allons montrer pourquoi ce modèle est en complète inadéquation avec l'analogie dossier/fichier. Ce cas simple permet met en évidence le problème : le modèle mixte fait tout simplement disparaître la belle logique des poupées russes.

Ces représentations graphiques montrent clairement que ce modèle fait apparaître un amalgame curieux entre le texte, les fichiers et les dossiers, le rôle de structuration hiérarchique des dossiers étant perdu et parfois même tenu par des objets de type fichier, les uns et les autres étant mêlés en un joyeux désordre ! On est loin de la représentation aisée du début de ce chapitre.

Tableau 9–5 Représentation des éléments mixtes

Exemple d'élément à contenu mixte

L'exemple utilisé ici est celui du cas précédent, auquel on a ajouté les balises `b` et `i` pour obtenir un contenu mixte :

```
<p>L'utilitaire <b>CATSTA</b> est utilisé pour traiter des données
statistiques et comptables sous <b><i>VM</i> et <i>MVS</i></b>.</p>
```

Il y a deux représentations possibles de ce fragment.

Tableau 9–5 Représentation des éléments mixtes

<table>
<tr><th>Représentation graphique par analogie avec un système de fichiers</th></tr>
<tr><td>Première représentation : on utilise des fichiers dont le nom est indépendant de celui des éléments.
<table>
<tr><td colspan="4">fichier.txt ⓿</td></tr>
<tr><td></td><td colspan="3"><p>L’utilitaire</td></tr>
<tr><td></td><td colspan="3">fichier.txt</td></tr>
<tr><td></td><td></td><td colspan="2">CATSTA</td></tr>
<tr><td></td><td colspan="3">est utilisé pour traiter des données statistiques et comptables sous</td></tr>
<tr><td></td><td colspan="3">un conteneur (représentant <b>) ❶</td></tr>
<tr><td></td><td></td><td colspan="2">fichier.txt</td></tr>
<tr><td></td><td></td><td></td><td><i>VM</i></td></tr>
<tr><td></td><td></td><td>et</td><td>❷</td></tr>
<tr><td></td><td></td><td colspan="2">fichier.txt</td></tr>
<tr><td></td><td></td><td></td><td><i>MVS</i></td></tr>
<tr><td></td><td colspan="3"></p></td></tr>
</table></td></tr>
<tr><td>Cette représentation fait apparaître de graves irrégularités dans la structure : présence d’un fragment textuel « flottant » (raccroché à rien) à l’intérieur d’un dossier (repère ❷) et d’un conteneur dont on ne sait ni spécifier le nom (repère ❶) ni expliquer comment il peut concrètement figurer dans un fichier (repère ⓿).</td></tr>
<tr><td>La seconde représentation s’appuie sur des dossiers et des fichiers auxquels on fait porter le nom de l’élément qu’ils représentent :
<table>
<tr><td colspan="2">p.txt</td><td colspan="2">⓿</td></tr>
<tr><td></td><td colspan="3">L’utilitaire</td></tr>
<tr><td></td><td colspan="3">b.txt</td></tr>
<tr><td></td><td></td><td colspan="2">CATSTA</td></tr>
<tr><td></td><td colspan="3">est utilise pour traiter des données statistiques et comptables sous</td></tr>
<tr><td></td><td>b</td><td colspan="2">❶</td></tr>
<tr><td></td><td></td><td colspan="2">i.txt</td></tr>
<tr><td></td><td></td><td></td><td>VM</td></tr>
<tr><td></td><td></td><td>et</td><td>❷</td></tr>
<tr><td></td><td></td><td colspan="2">i.txt</td></tr>
<tr><td></td><td></td><td></td><td>MVS</td></tr>
</table>
Cette représentation montre que les problèmes de la chaîne flottante (repère ❷) et du conteneur b (repère ❶) contenu dans le fichier p (repère ⓿) sont toujours présents.</td></tr>
</table>

Cela ne pourrait fonctionner que si l'on savait imbriquer des dossiers dans des fichiers, des fichiers dans des fichiers, et stocker du texte directement dans des dossiers ! Le seul moyen de réaliser physiquement un tel modèle logique serait d'avoir la possibilité de poser des liens allant de l'intérieur du document vers des dossiers qui contiendraient des fichiers… Notre exemple montre à quel point cela serait fastidieux à mettre en œuvre et surtout incohérent pour la rédaction d'un tel petit texte. Ce modèle est en réalité très proche du modèle HTML, dont on connaît les graves lacunes de structuration.

Souvenons-nous du problème essentiel mis en évidence avec le modèle mixte : dans notre exemple, on ne sait pas à quoi raccrocher la chaîne de caractères `et` (repère ❷). Ce problème existe en HTML et c'est uniquement la permissivité des navigateurs qui a masqué le problème : ceux-là considèrent que toute chaîne de caractères flottante est par défaut encadrée d'un élément `p` de type paragraphe. Si cela est acceptable quand il s'agit simplement d'un problème d'affichage dans un navigateur, ça ne l'est plus quand il s'agit d'être rigoureux sur la structure de l'information, notamment eu égard à des bases de données. Nous allons voir à quel point ces anomalies peuvent provoquer de sérieux blocages si la règle de l'élément `p` induit devenait réelle. Appliquée au fragment précédent, cela donnerait :

```
<p>L'utilitaire <b>CATSTA</b> est utilisé pour traiter des données
statistiques et comptables sous <b><i>VM</i> <p>et</p> <i>MVS</i>
</b>.</p>
```

Et la présence incongrue de cet élément `p` est non seulement absurde mais également invalide (à moins de faire un modèle récursif où `p` peut se contenir lui-même…).

Ainsi, toute l'information se trouvant à l'intérieur d'un élément mixte doit être considérée comme étant de la donnée non structurée, et l'élément lui-même doit être *un élément de données*.

C'est pour cela que nous disons que la forme en poupées russes du modèle XML est moins régulière qu'il n'y paraît de prime abord. Dans ce chapitre, nous développons l'idée que, dans un document XML, tous les éléments ne sont pas structurellement égaux.

Cas des éléments vides

En XML, les éléments vides finissent souvent par avoir un contenu. Par exemple, une balise d'appel de figure est remplacée par ladite figure au moment de la composition du document. Il en est de même pour toutes les balises formant des références croisées. Lors de la composition, elles sont remplacées par du contenu textuel de type « *voir aussi*…. », « *voir page*…. ». Il existe de nombreux autres cas de figure.

On aime donc mieux considérer les éléments vides comme des objets prêts à recevoir potentiellement un contenu plutôt que comme des objets nuls. Les éléments vides sont pour nous des éléments de type simple dont le contenu est ramené à un ensemble vide : c'est-à-dire des fichiers vides.

Nous disons que ces éléments sont *des éléments de données*.

Où doit-on mettre les attributs ?

À ce stade, notre raisonnement n'a pas tenu compte des attributs XML.

En XML Schema, la présence d'attributs ne fait pas le type d'un élément. Il n'y a que deux types d'éléments : simple et complexe. Les éléments de type simple ont un contenu limité à du texte et aucun attribut. Tous les autres sont de type complexe.

Cela indique que les concepteurs de XML Schema ont voulu rapprocher les éléments de type simple des attributs. Du point de vue du typage, il n'y a pas de différence entre les deux. Du point de vue conceptuel, écrire `<info><date>23/05/59</date></info>` revient à écrire `<info date="23/05/59"/>`.

En XML, les attributs sont clairement dégagés de toute responsabilité structurelle : ils sont dans une dimension perpendiculaire à elle. Les attributs sont des propriétés placées sur les éléments et un ensemble d'attributs forme une fiche de propriétés.

PARTICULARITÉ DE XML SCHEMA **Le typage dynamique des éléments**

Avec XML Schema, il est possible dans un document XML de spécifier à la volée le type d'un élément. Il suffit pour cela d'utiliser sur l'élément concerné l'attribut `xsi:type` : la seule condition est de spécifier le nom d'un type qui soit un dérivé valide du type originel de l'élément. Ce mécanisme est toutefois trop restrictif pour qu'on puisse dire que XML Schema permet d'influer sur le modèle de contenu ou type d'un élément à partir de la valeur d'un attribut. Par exemple, il est strictement impossible de définir des règles telles que : « Si la valeur de l'attribut date de l'élément achat est supérieure à 2002, alors les sous-éléments autorisés sont euros et cents tandis que, si cette même valeur est inférieure à 2002, les sous-éléments autorisés sont francs et centimes. »

Dans la suite de ce chapitre, nous allons utiliser de nombreuses représentations graphiques de documents épurées du nom des attributs. Nous irons même jusqu'à supprimer le nom des éléments car, dans le cadre du sujet de ce chapitre, seule la forme du document importe. Ainsi, le document que nous fournissons ci-après sera représenté par la figure 9-2 et non par la figure 9-1.

```
<piloteWeb xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="sites.xsd">
<sites>
<site unique-id="ID000000">
   <nom value="Nicolas le Panda"/>
   <identification>
    <coordonnées>
     <position>
      <longitude value="48.7494"/>
      <latitude value="-2.1108"/>
     </position>
     <altitude value="900"/>
    </coordonnées>
   </identification>
   <photo nom="photo.gif" format="gif"/>
  </site>
</sites>
</piloteWeb>
```

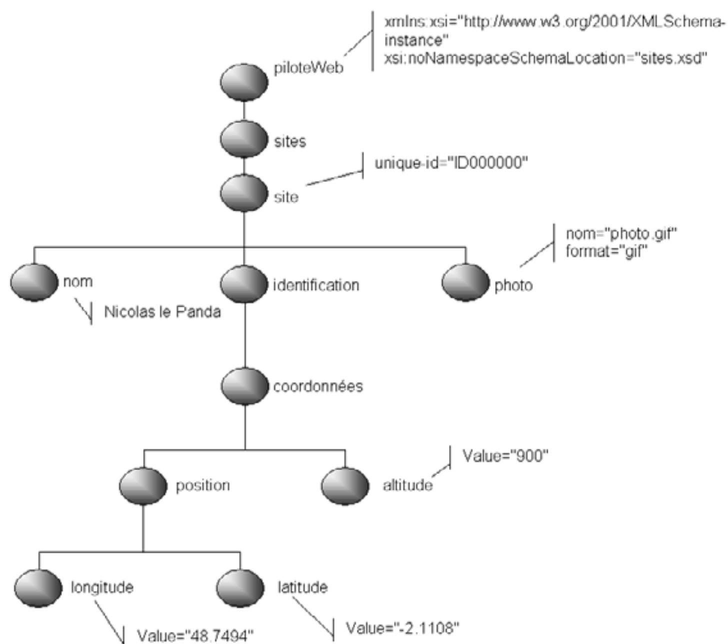


Figure 9–1
Représentation graphique complète (éléments + attributs) d'un document XML

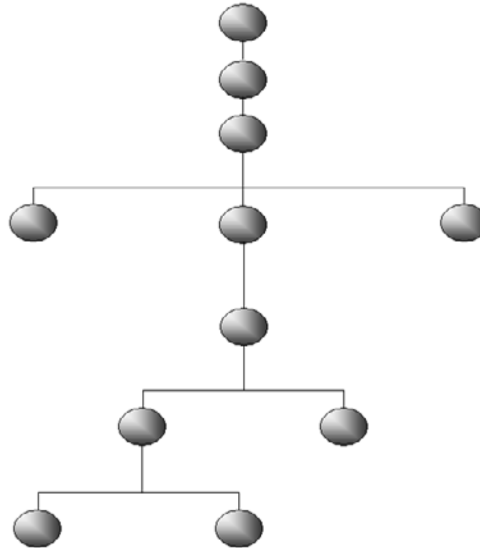


Figure 9–2
Représentation graphique simplifiée d'un document XML

Que faire des métadonnées ?

Comme leur nom l'indique, les métadonnées sont des données sur les données. Leur rôle est donc de qualifier les autres données du document. Cela n'a toutefois aucun rapport avec le terme de métadonnées utilisé dans les bases de données. Cette précision est importante car XML étant utilisé indifféremment dans les deux mondes – celui des documents et celui des bases de données –, il y a un risque certain de confusion. Dans le monde des documents, une métadonnée apporte, à la manière des attributs, une information qui n'est pas limitée à la description du type des autres données. Il s'agit par exemple d'informations de gestion, de provenance, de statut, etc., applicables à tout ou partie d'un document. Dans le monde des bases de données, les métadonnées indiquent le type des données.

Par exemple, l'élément `meta` de HTML est une métadonnée. Il se trouve dans l'élément `head`, autorisé en début de page HTML. Bien que frère de l'élément `body`, l'élément `head` contient des données descriptives des données contenues dans `body`. Dans ce cas particulier, les éléments `meta` forment la fiche de propriétés de l'élément `body` : ils qualifient un sous-ensemble complet d'informations.

D'une façon générale, les métadonnées d'un document XML ont les particularités suivantes :

- Elles qualifient l'ensemble du document XML. À ce titre, les éléments contenant les métadonnées sont comme des attributs qui s'appliqueraient à la totalité du document XML.

- Les éléments contenant les métadonnées sont généralement réunis sous un même élément racine. Ils forment un ensemble d'informations assimilable à une fiche d'identité qui pourrait servir d'étiquette au document XML et être stockée à l'extérieur du document XML.

Une conséquence très importante de ces deux points est que les éléments porteurs de métadonnées doivent être assimilés, du point de vue structurel, à des attributs. Comme eux, on doit considérer qu'ils font partie d'un plan orthogonal à celui de la structure. Comme nous le laissons entendre dans le second point susmentionné, les métadonnées doivent pouvoir être indifféremment stockées à l'intérieur ou à l'extérieur du document XML.

Pour donner des exemples concrets de métadonnées, nous proposons quelques utilisations de la balise `meta` de HTML :

```
<html>
 <head>
  <title>Bash Reference Manual: </title>
  <meta name="description" content="Bash Reference Manual: ">
  <meta name="keywords" content="Bash Reference Manual: ">
  <meta name="resource-type" content="document">
  <meta name="distribution" content="global">
  <meta name="Generator" content="texi2html 1.64">
 </head >
<body>
… … … … …
</body>
```

Ou encore :

```
<html xmlns:o="urn:schemas-microsoft-com:office:office"
      xmlns:w="urn:schemas-microsoft-com:office:word"
      xmlns="http://www.w3.org/TR/REC-html40">
 <head>
  <meta http-equiv="Content-Type"
        content="text/html; charset=windows-1252">
  <meta name="ProgId" content="Word.Document">
  <meta name="Generator" content="Microsoft Word 9">
  <meta name="Originator" content="Microsoft Word 9">
  <link rel="File-List" href="./temp_files/filelist.xml">
  <title>La boîte à outils de base</title>
</head>
<body>
… … … … …
</body>
```

Dans ces fragments, les informations portées par la balise `meta` servent à qualifier le corps (`body`) de la page HTML. Ces métadonnées agissent comme une série d'attributs portée par l'élément racine `html`. S'il n'a pas été procédé ainsi, c'est parce que l'élément racine, comme tout élément, ne peut pas porter plusieurs attributs de même nom. Les attributs `name` et `content` de la balise `meta` ne peuvent pas être répétés sur un même élément, fût-il l'élément racine. On ne peut donc pas faire autrement que d'utiliser des éléments `meta` répétés à l'intérieur de l'élément `head`. Cet ensemble d'information agit comme si toutes ces données étaient portées par le seul élément `html`.

Ces données représentent typiquement une information contenue dans le document, mais qui ne devrait pas l'être ! Nous aurons l'occasion de montrer comment ces informations, qui devraient être stockées à l'extérieur du document, viennent souvent perturber la représentation idéale d'un document XML par un arbre constitué, dans ses premiers niveaux, d'une structure régulière de dossiers emboîtés les uns dans les autres. Les métadonnées nécessaires à la gestion des documents XML perturbent ce bel édifice.

Les éléments XML ne sont pas tous structurellement égaux

Ce qui peut passer ici pour des subtilités est pourtant nécessaire à la compréhension de la théorie des éléments purement structurels. Au terme de cette analyse préliminaire, peut-on encore dire que tous les éléments XML sont égaux ? La réponse est non.

En XML, le système des poupées russes n'est pas régulier, et nous avons vu que cela tient autant de la nature du langage XML que de celle des données.

Non seulement les éléments XML sont inégaux sur le plan structurel, mais ils le sont aussi sur le plan fonctionnel. Par exemple, une balise de mise en gras joue un rôle important quand il s'agit d'afficher un document (conséquence visible), mais cette même balise encadrant un prix n'apporte rien à une application comptable. En contrepartie, les éléments qui ont de l'importance pour une application de gestion électronique de documents sont ceux qui servent à le gérer : les éléments racines, les métadonnées… Plus l'élément est proche de la donnée et plus son importance est grande pour les applications qui la manipulent ; plus il est structurel et plus son rôle devient important pour la gestion de l'information :

- Les fonctions de gestion (pose de métadonnées, découpage du document XML, workflow…) ne sont applicables qu'aux seuls éléments purement structurels.
- Toute tentative de gestion sur un autre type d'élément est vaine.

La méthode exposée dans la suite de ce chapitre aide à découvrir les éléments purement structurels d'un schéma. Seuls ces éléments permettent de découper un document XML en plusieurs sous-fichiers en vue d'instituer un modèle de gestion avec

des petits modules indépendants, comme si le document XML était découpé en plusieurs petits fichiers. Gérer l'information plus finement qu'au seul niveau de l'élément racine est impératif quand les documents XML deviennent trop gros ou résultent de l'assemblage de petits morceaux produits en parallèle : il n'est alors pas concevable de se contenter du seul niveau fichier.

Découvrir les éléments purement structurels

Règles d'identification

Cette section présente les règles formelles qui aident à découvrir les éléments purement structurels d'un schéma XML.

Dans la recherche des éléments purement structurels, il y a deux situations radicalement différentes :

- La qualité d'élément purement structurel a été prise en compte au niveau du schéma par l'écriture d'un modèle adapté, et tous les éléments d'un nom donné auront cette qualité.
- La qualité d'élément purement structurel ne se dévoile que dans les documents XML, en fonction du contexte d'utilisation des éléments autorisés par le schéma. Ces éléments sont instables, tantôt purement structurels, tantôt simples éléments de données. La méthode que nous présentons permet de les identifier.

Ainsi, les éléments purement structurels sont tantôt définis formellement par le schéma et tantôt dépendant d'aléas d'utilisation. Seules les applications dont les éléments purement structurels sont clairement définis par les schémas XML permettent d'envisager une gestion modulaire saine des données.

Les règles servant à identifier les éléments purement structurels utilisent les expressions déjà vues d'*élément de données*, *élément de données global* et *élément purement structurel*, dont voici les définitions formelles :

- Un *élément de données* définit tout élément dont le modèle de contenu est de type simple ou mixte. Tout élément qui n'est pas purement structurel est un élément de données.
- Un *élément de données global* est le premier *élément de données* rencontré dans une branche de l'arbre quand on le lit dans le sens hiérarchique de la racine aux feuilles.
- Un *élément purement structurel* ne contient aucune séquence d'élément de données comme enfant direct. Une séquence est une série de deux éléments ou plus.

La qualité d'élément purement structurel ne se décrète pas par une définition ; elle se découvre par nettoyage progressif d'un schéma en appliquant les six règles déclinées ci-après, selon une méthode qui sera vue dans une section suivante. Ce n'est qu'une fois ce travail d'épuration réalisé qu'un schéma dévoile ses éléments purement structurels.

- **Règle n° 1 ou règle d'absorption.** Tous les éléments enfants d'un élément de données sont eux-mêmes des éléments de données.
- **Règle n° 2 ou règle de l'enfant unique.** Un élément dont le seul enfant est un élément de données est un élément purement structurel.
- **Règle n° 3 ou règle de la promotion.** Un élément dont les enfants sont tous des éléments purement structurels ou des éléments de données globaux est lui-même un élément purement structurel.
- **Règle n° 4 ou règle de la mixité.** Un élément contenant un ensemble mêlant éléments de données et éléments purement structurels ne peut pas être transformé en élément purement structurel. Cette règle s'applique quand les connecteurs `xs:all` ou `xs:choice` sont constitués d'éléments de données et d'éléments purement structurels ou quand les enfants d'un élément mixte sont purement structurels. Ce dernier cas rejoint la règle n° 1.
- **Règle n° 5 ou règle de la tutelle.** Si un élément contient une séquence d'éléments de données et d'éléments purement structurels, le seul moyen de transformer cet élément en élément purement structurel est de faire chapeauter tous les éléments de données par un nouvel élément.
- **Règle n° 6 ou règle de l'élément célibataire.** Un élément vide est un élément de données.

REMARQUE Élément de données global

La notion d'élément de données global n'est intéressante que pour mettre en évidence les nœuds hybrides entre données et structure. Pour la seule recherche des éléments purement structurels, les éléments de données globaux n'apportent rien de particulier par rapport aux simples éléments de données, et d'ailleurs ils n'interviennent pas dans les six règles énoncées.

Exemples d'application

Pour illustrer les règles susmentionnées d'identification des éléments purement structurels, nous allons utiliser une représentation graphique sous forme d'arbre d'un document XML.

Pour les raisons expliquées plus haut, l'exemple de la figure 9-3 ne mentionne ni les attributs ni le nom des éléments. C'est un cas assez particulier puisqu'il ne contient aucun élément de données à proprement parler. Tous les éléments sont vides ou contiennent des sous-éléments. Rappelons-nous toutefois que la règle n°6 conduit à

considérer les éléments vides comme des éléments de données. Commençons donc par les mettre en évidence en les grisant (repères ❷, ❹, ❼, ❽ et ❾ de la figure 9-3).

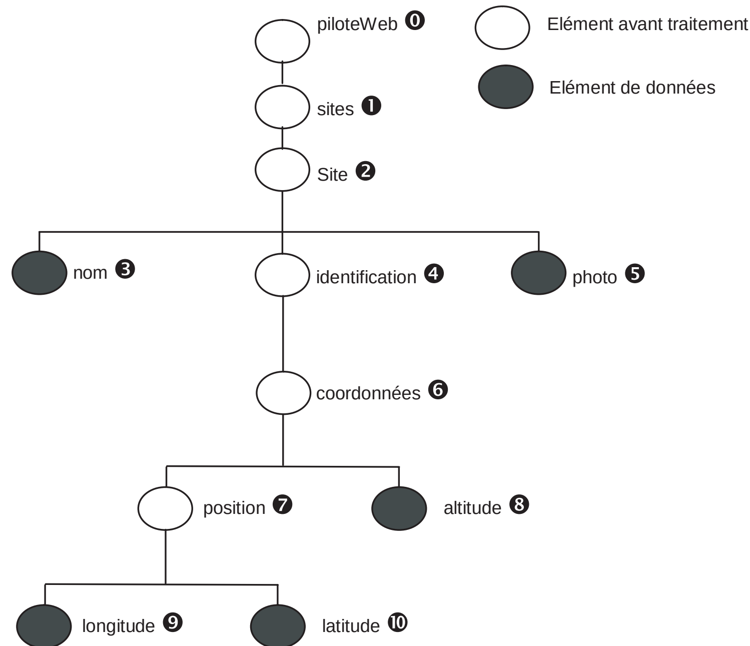


Figure 9–3
Exemple de graphe initial avant recherche des éléments purement structurels

Nous allons appliquer les règles édictées en commençant par les extrémités de l'arbre les plus éloignées de la racine.

Appliquons la définition d'un élément purement structurel au nœud ❻. Ce nœud contient deux éléments de données, il n'est pas purement structurel. Nous le grisons, et notre arbre devient celui illustré à la figure 9-4.

Cet élément devient le premier élément de données global de cette sous-branche.

Suite à cette modification, on met en évidence que son parent immédiat (l'élément ❺) n'est pas purement structurel puisqu'il contient une série de deux éléments de données, l'un global, l'autre simple. Nous le grisons donc à son tour. L'élément parent de celui que nous venons de griser, le ❸, est le premier élément purement structurel que nous rencontrons parce qu'il n'a qu'un seul enfant (règle n°2) ; cela donne la situation illustrée par la figure 9-5.

Par conséquent, nous avons sous l'élément ❶ une série composée d'un élément de données ❷, d'un élément purement structurel ❸ et à nouveau d'un élément de données ❹. La règle n°4 indique que cet élément ne peut être transformé en un élément purement structurel. Nous sommes donc tenus de considérer que c'est un élé-

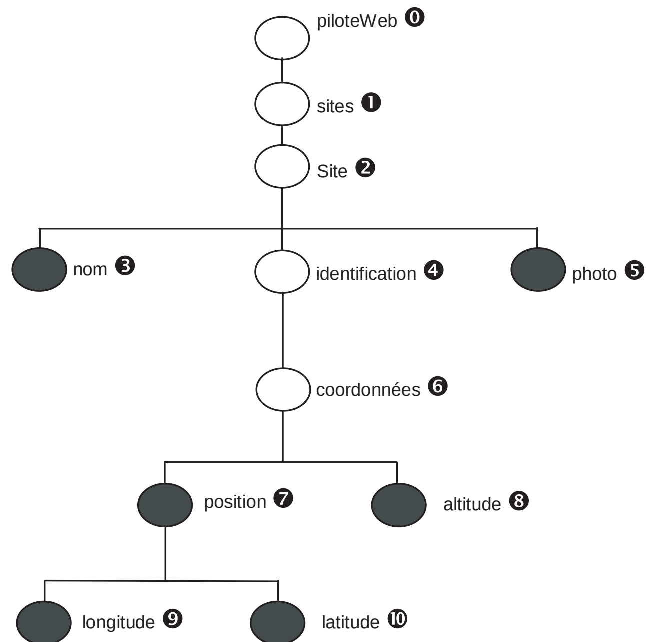


Figure 9–4
Graphe après analyse du nœud 7

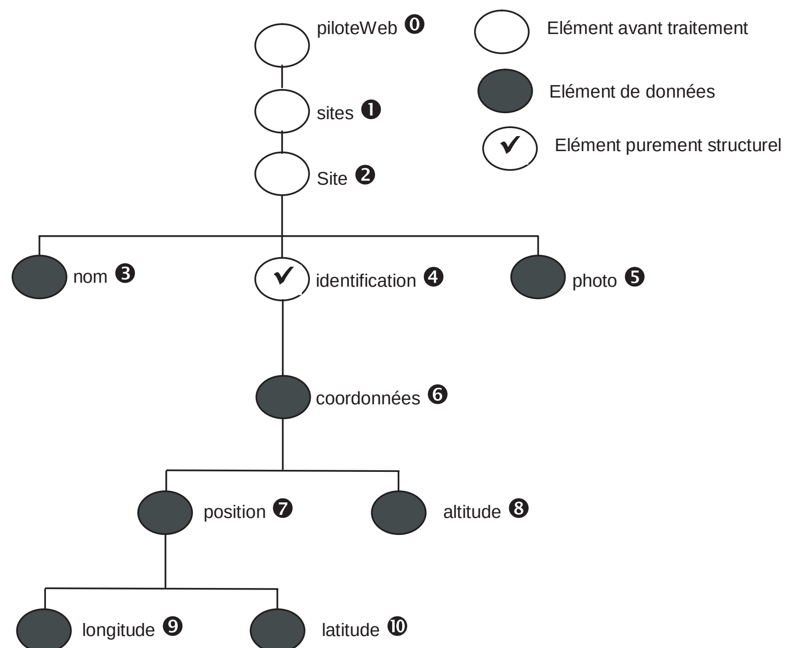
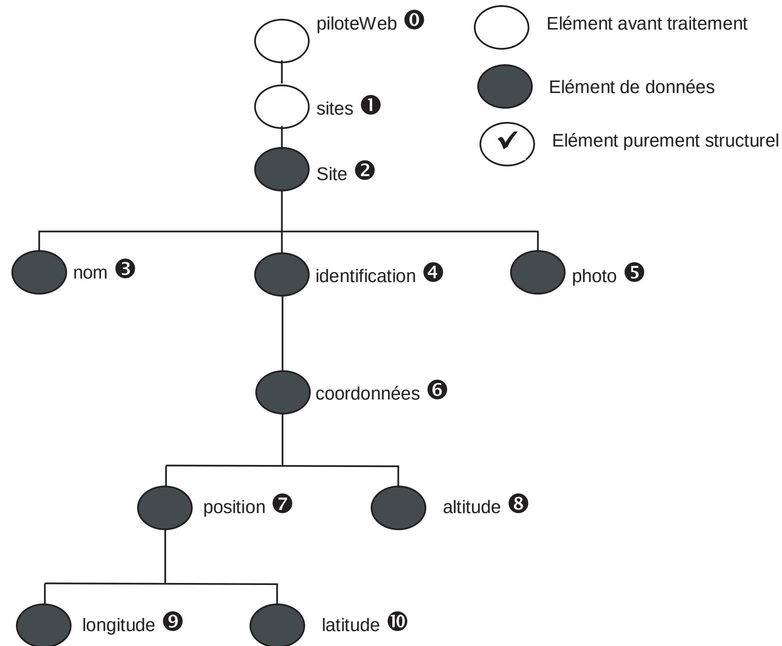


Figure 9–5
Graphe après analyse du nœud 6

ment de données, et nous le grisons. Nous appliquons alors la règle n°1 et grisons l'élément purement structurel ❸ que nous venons de trouver, puisque nous ne pouvons pas avoir d'élément purement structurel sous un élément de données. On dit qu'il est absorbé.

Le schéma résultant est illustré à la figure 9-6.

Figure 9–6
Graphe après analyse des nœuds 2 et 4



Nous pouvons finalement appliquer la règle n°2 à l'élément ⓿ et dire qu'il est purement structurel, de même que son parent ❿, en vertu de la règle n°3, ce qui donne le résultat illustré à la figure finale 9-12.

Au terme de l'analyse de ce fragment XML, on a pu déterminer que seuls les deux premiers éléments sont purement structurels. Par conséquent, le seul endroit possible pour découper ce fragment en modules se situe entre les éléments ❿ et ⓿. En d'autres termes, et pour reprendre notre analogie de début de chapitre, les deux seuls éléments assimilables à des dossiers sont les éléments ❿ et ⓿, et l'élément ❶ est obligatoirement du niveau fichier.

VOCABULAIRE PSE

L'expression *élément purement structurel* se traduit en anglais par *Pure Structural Element*, dont le sigle est PSE. Nous utiliserons dans la suite de ce chapitre le sigle PSE au lieu de l'expression française complète.

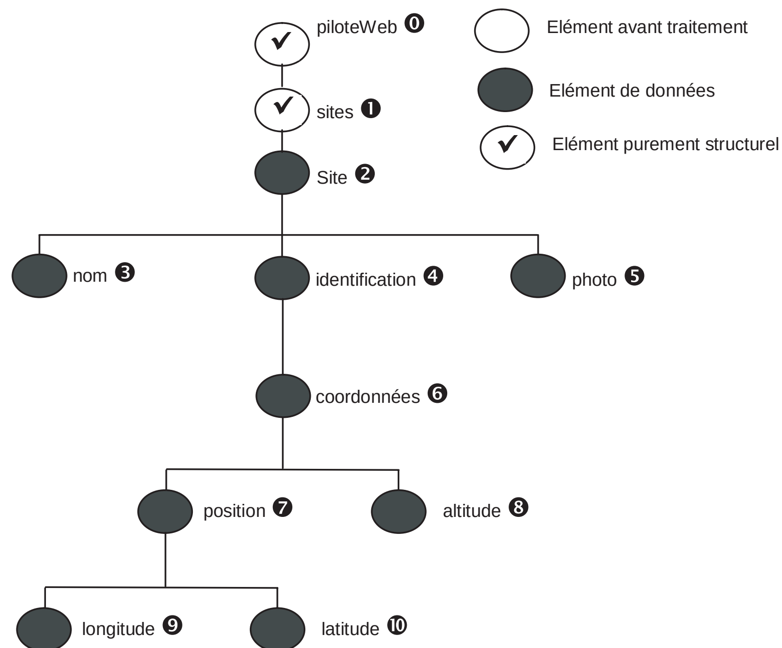


Figure 9–7
Graphe après analyse des nœuds 0 et 1

Méthode pratique d'application des règles

S'appuyant sur un cas concret, cette section détaille la méthode qu'il convient de suivre pour mettre en évidence les éléments purement structurels d'un schéma.

Le schéma utilisé ici comme exemple, connu sous le nom de J2008, a été conçu pour un métier bien spécifique : celui des équipementiers de l'automobile. Les noms donnés aux éléments XML sont le reflet de ce métier. Pour mettre en évidence les éléments purement structurels d'un schéma, il n'est cependant nul besoin d'avoir la connaissance d'un métier et d'en comprendre le vocabulaire. La méthode ne s'appuie que sur la forme du modèle XML défini par le schéma.

Nous avons commencé par retirer du schéma toutes les déclarations d'attributs et de notations, qui ne servent à rien dans la recherche des PSE.

Le schéma que nous utilisons ici comme exemple fait mille six cents lignes. Pour trouver les éléments purement structurels qu'un modèle de ce type contient, il n'existe pas d'autre solution que de mettre en œuvre la méthode que nous proposons. Sur un schéma aussi important, la mise à jour des PSE prend environ une heure.

La méthode comporte 12 étapes. Pour chacune, nous présentons la règle à appliquer et le résultat attendu.

La méthode a été conçue de façon qu'elle soit appliquée manuellement, mais elle pourrait aussi bien faire l'objet d'un programme, lequel n'existe pas à ce jour. Pour la dérouler, nous avons utilisé le logiciel XML Spy et un éditeur de fichiers ASCII.

Étape 1. Préparation de l'environnement

Cette première étape consiste à préparer l'environnement de travail. Si vous n'avez pas de schéma XML mais seulement une DTD, convertissez-la en utilisant la fonction Convert DTD/Schema de XML Spy. La méthode que nous exposons, bien que facile à transposer aux DTD de XML 1.0, n'est expliquée que par rapport à la syntaxe des schémas XML du W3C. Nous recommandons d'utiliser les options de XML Spy illustrées à la figure 9-8.

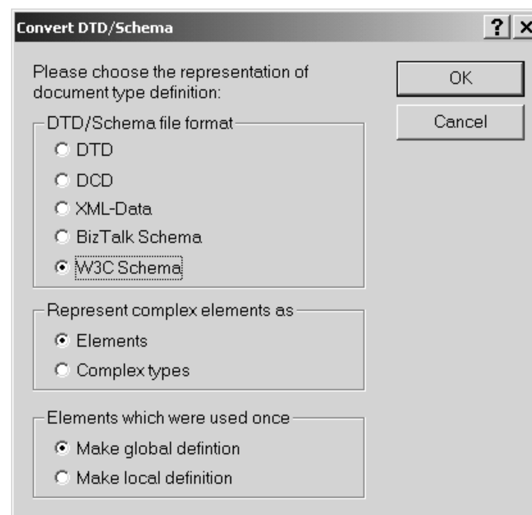


Figure 9–8
Options de XML Spy à utiliser pour la conversion DTD vers XML Schema

Si vous partez d'une DTD, la conversion en XML Schema ne peut se faire que si cette DTD est valide par rapport à XML 1.0, la conversion ne marchant pas pour des DTD SGML.

Très important : faites des copies de vos DTD et schémas originels. Les étapes que nous allons suivre détruisent les fichiers de travail.

Étape 2. Supprimer du schéma les commentaires et toutes les définitions d'attributs ou de notations

Il faut supprimer du schéma toutes les unités d'information de type `xs:attribute` et `xs:notation` :

```
Exemples : Les lignes suivantes ont été supprimées de notre schéma :
<xs:notation name="cgm" public="-//USA-DOD//NOTATION Computer Graphics
Metafile//EN"/>
<xs:notation name="cgmbin" public="ISO 8632:1993//NOTATION Binary
encoding//EN"/>
…
<xs:attribute name="configgrpSGMLid" type="xs:ID" use="required"/>
<xs:attribute name="mcseqnbr" type="xs:NMTOKEN" use="required"/>
<xs:attribute name="bodycabmfrcode" type="xs:NMTOKEN" use="required"/>
<xs:attribute name="bodycabdesc" type="xs:string" use="required"/>
<xs:attribute name="nbrofdoors" type="xs:NMTOKEN" use="required"/>
…
```

REMARQUE Si vous partez d'une DTD

Si vous partez d'une DTD, nous vous recommandons de faire cette suppression avant la conversion de la DTD en schéma XML. La suppression des commentaires n'est pas obligatoire.

Étape 3. Création de trois éléments vides, SDE, GDE et PSE

Cette étape consiste à créer les trois éléments dont nous aurons besoin dans les prochaines étapes :

- SDE (Simple Data Element), élément de donnée ;
- GDE (Global Data Element), élément de donnée global ;
- PSE (Pure Structural Element), élément purement structurel.

Mettez ces trois définitions en début de schéma, en reprenant les lignes fournies ci-après :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
 <xs:element name="SDE">
  <xs:complexType/>
 </xs:element>
 <xs:element name="GDE">
  <xs:complexType/>
 </xs:element>
```

```
<xs:element name="PSE">
 <xs:complexType/>
</xs:element>
```

Étape 4. Remplacement de tous les éléments ayant un contenu mixte par l'élément SDE

Pour cela, il faut exécuter la recherche illustrée à la figure 9-9.

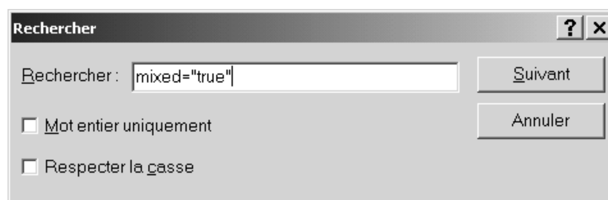


Figure 9–9
Recherche des éléments à contenu mixte d'un schéma

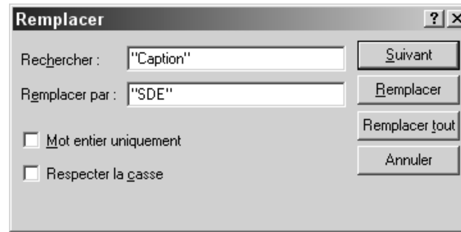
Cette recherche permet de trouver des structures de ce type :

```
<xs:element name="Caption">
 <xs:complexType mixed="true"> ← l'information recherchée
  <xs:choice minOccurs="0" maxOccurs="unbounded">
   <xs:element ref="Emph"/>
   <xs:element ref="Sub"/>
   <xs:element ref="Sup"/>
   <xs:element ref="Ftnote"/>
   <xs:element ref="Intxref"/>
   <xs:element ref="Figureref"/>
   <xs:element ref="Tableref"/>
   <xs:element ref="Diagref"/>
   <xs:element ref="extxref"/>
   <xs:element ref="Symbol"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

Vous allez maintenant appliquer les modifications suivantes à ces structures :

1. Supprimez la totalité du contenu compris entre les balises (incluses) `<xs:complexType>` et `</xs:complexType>`.
2. Faites un remplacement global du nom de l'élément concerné (`Caption` dans notre cas) par `SDE`, comme illustré à la figure 9-10.

Figure 9–10
Remplacement des éléments mixtes par des SDE



> **REMARQUE Attention aux guillemets**
>
> Il est important d'utiliser systématiquement les guillemets anglais lors de ces remplacements. Sinon, vous prenez le risque de remplacer définitivement et de manière imprévue des chaînes de caractères, ce qui vous obligerait à recommencer le processus au début.

Le résultat obtenu pour votre fragment doit être :

```
<xs:element name="SDE">
</xs:element>
```

Le mot `Caption` (parce que c'est ici le nom de la balise de notre exemple) doit avoir été remplacé dans tous les modèles de contenu où il était référencé :

```
<xs:sequence>
<xs:element ref="Graphic"/>
<xs:element ref="SDE" minOccurs="0"/>
</xs:sequence>
```

Vous pouvez dès lors supprimer la définition originelle de l'élément `Caption`, dont vous n'avez plus besoin, en supprimant les deux lignes suivantes :

```
<xs:element name="SDE">
</xs:element>
```

Répétez le processus autant de fois qu'il y a de contenus mixtes et faites une sauvegarde du fichier à la fin de cette étape. On entend par sauvegarde une véritable copie du fichier, pour éviter de devoir recommencer le processus au début en cas d'erreur ultérieure.

Étape 5. Remplacement de tous les éléments vides par l'élément SDE

Cette étape est, dans son principe, identique à la précédente : elle consiste à rechercher les éléments vides qui se caractérisent par un type complexe vide. Pour les trouver, exécutez la recherche illustrée à la figure 9-11.

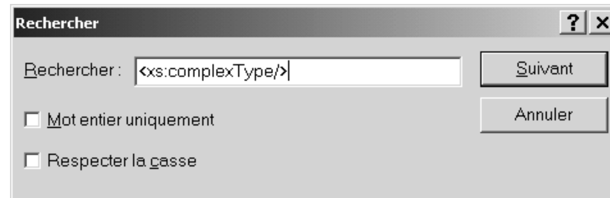


Figure 9–11 Recherche des types complexes vides

Voici un exemple de fragment trouvé :

```
<xs:element name="Topicref">
 <xs:complexType/> ← l'information recherchée
</xs:element>
```

Comme pour l'étape précédente, la méthode consiste à :

1. Remplacer globalement le nom de l'élément vide par SDE ; dans notre exemple, il faut remplacer toutes les occurrences de `Topicref` par `SDE`.
2. Supprimer la définition originelle de l'élément. Dans notre exemple, il faut supprimer les trois lignes de la définition de l'élément `Topicref`.

Faites de nouveau une sauvegarde du fichier à la fin de cette étape.

Étape 6. Remplacement de tous les éléments simples par l'élément SDE

La procédure est toujours la même : rechercher les éléments de type simple et remplacer les noms des éléments trouvés par le mot `SDE`.

Les définitions d'éléments de type simple contiennent les attributs `name` et `type`. Elles sont de la forme :

```
<xs:element name="Sub" type="xs:string"/>
<xs:element name="Sup" type="xs:string"/>
```

Dans cet exemple, il faut faire un remplacement global de `Sub` et `Sup` par `SDE`, puis détruire ces définitions.

Faites une sauvegarde de votre schéma à la fin de cette étape.

Étape 7. Transformation en GDE des éléments dont le modèle de contenu est une série d'éléments SDE

Il s'agit maintenant de détecter les éléments dont le modèle de contenu n'est composé que d'une séquence d'éléments SDE.

Il peut s'agir :

- d'un modèle composé d'un élément SDE répété n fois, par exemple :

```
<xs:complexType name="VehConfigVarYrType">
 <xs:sequence>
  <xs:element ref="SDE" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
```

- d'une séquence d'éléments SDE :

```
<xs:element name="Para">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="SDE" minOccurs="0"/>
   <xs:element ref="SDE"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

- d'un choix non répétitif entre des éléments SDE :

```
<xs:element name="EngineMotor">
  <xs:complexType>
   <xs:choice>
    <xs:element ref="SDE"/>
    <xs:element ref="SDE"/>
   </xs:choice>
  </xs:complexType>
 </xs:element>
```

- d'un choix répétitif entre des éléments SDE :

```
<xs:element name="ListofSIEs">
 <xs:complexType>
  <xs:choice maxOccurs="unbounded">
   <xs:element ref="SDE"/>
```

```
    <xs:element ref="SDE"/>
   </xs:choice>
  </xs:complexType>
</xs:element>
```

Ces éléments étant des éléments de données globaux, il faut les remplacer par autant d'éléments GDE, puis supprimer, comme précédemment, leurs définitions originelles au fur et à mesure que les remplacements sont effectués.

Faites une sauvegarde de votre schéma à la fin de l'opération.

Étape 8. Transformation en PSE des éléments dont le modèle de contenu est une série d'éléments GDE

Cette étape consiste à détecter les éléments dont le modèle de contenu n'est composé que d'une séquence d'éléments GDE.

Il peut s'agir :

- d'un élément GDE répété n fois :

```
<xs:element name="VehicleVars">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="GDE" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

- d'une séquence d'éléments GDE :

```
<xs:element name="Para">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="GDE"/>
   <xs:element ref="GDE"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

- d'un choix entre éléments qui sont tous des GDE :

```
<xs:element name="Para">
 <xs:complexType>
```

```
  <xs:choice>
   <xs:element ref="GDE"/>
   <xs:element ref="GDE"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

Ces éléments étant des éléments purement structurels, il faut les remplacer par autant d'éléments PSE, puis supprimer, comme précédemment, leurs définitions originelles au fur et à mesure que les remplacements sont effectués.

Faites une sauvegarde de votre schéma à la fin de l'opération.

REMARQUE PSE : conserver le nom de l'élément originel

Nous recommandons pour les éléments PSE de conserver le nom de l'élément originel (en écrivant, par exemple, PSE-Para), sans quoi il serait impossible de les retrouver une fois la méthode achevée, ce qui serait exactement contraire au but recherché.

Étape 9. Remplacement de tous les éléments qui n'ont qu'un seul enfant possible

Ce remplacement dépend du type de l'élément enfant et de la valeur des indicateurs d'occurrences.

Le tableau 9-6 recense les différents cas de figure et indique, pour chacun, ce par quoi il faut remplacer l'élément concerné.

Tableau 9–6 Tableau 9-6. Remplacement des éléments qui n'ont qu'un seul enfant

Type de l'élément enfant	Valeur des indicateurs d'occurrences	Type à donner à l'élément parent
SDE	0,n	SDE (parce que le père est potentiellement vide).
	1,n	GDE
	1,1	GDE, PSE autorisé
GDE	0,n	SDE (parce que le père est potentiellement vide).
	1,n	PSE
	1,1	PSE
PSE	0,n	SDE (parce que le père est potentiellement vide).
	1,n	PSE
	1,1	PSE

Étape 10. Remplacement en GDE de tous les connecteurs xs:choice qui contiennent des éléments SDE mêlés à d'autres

Ne pouvant présager du choix qui sera fait par l'utilisateur du schéma, tout élément dont le modèle de contenu est un choix comprenant au moins un élément SDE doit être transformé en GDE. C'est la règle n°4 qui s'applique. La présence d'un seul élément de données dans un choix empêche de déclarer l'élément parent comme étant purement structurel, même si tous les autres éléments du choix sont des éléments purement structurels ou des éléments de données globaux.

D'une manière générale, on retient l'option minimaliste. Le type de l'élément parent est toujours dicté par l'élément de moindre qualité structurelle du modèle de contenu.

Le cas des éléments n'ayant qu'un seul enfant dans l'instance est abordé à l'étape suivante.

Voici un exemple :

```
<xs:element name="Condition">
 <xs:complexType>
  <xs:choice maxOccurs="unbounded">
   <xs:element ref="GDE"/>
   <xs:element ref="Spec"/>
   <xs:element ref="GDE"/>
   <xs:element ref="GDE"/>
   <xs:element ref="GDE"/>
   <xs:element ref="GDE"/>
   <xs:element ref="GDE"/>
   <xs:element ref="GDE"/>
   <xs:element ref="Paragroup"/>
   <xs:element ref="SDE"/>
   <xs:element ref="GDE"/>
   <xs:element ref="SDE"/>
   <xs:element ref="Figure"/>
   <xs:element ref="SDE"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

Dans ce modèle de contenu, les éléments sont pour la plupart de type GDE. Bien qu'il reste encore trois éléments de type structurel inconnu (Spec, Paragroup et Figure), trois éléments de type SDE sont présents. Cela suffit pour décider que l'élément Condition est de type GDE, alors qu'on aurait pu le croire de type PSE.

Résultat final

L'application de cette méthode sur le schéma J2008 aboutit au résultat suivant, dans lequel ont été supprimées les définitions inutiles d'éléments SDE, GDE et PSE, mais où ont été conservées toutes les déclarations d'éléments purement structuraux ainsi que les éventuels GDE contenant des PSE :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
 <xs:element name="PSE-J2008">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="GDE" minOccurs="0"/>
    <xs:element ref="PSE-ServInfoPool" minOccurs="0"/>
    <xs:element ref="PSE-OEMinfo" minOccurs="0"/>
    <xs:element ref="PSE-paths" minOccurs="0"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="PSE-OEMinfo">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="GDE" minOccurs="0"/>
    <xs:element ref="PSE-VehicleVars" minOccurs="0"/>
    <xs:element ref="GDE" minOccurs="0"/>
    <xs:element ref="GDE" minOccurs="0"/>
    <xs:element ref="GDE" minOccurs="0"/>
    <xs:element ref="GDE" minOccurs="0"/>
    <xs:element ref="PSE-ConfigVars" minOccurs="0"/>
    <xs:element ref="PSE-ConfigVarYrs" minOccurs="0"/>
    <xs:element ref="PSE-VehConfigVarYrs" minOccurs="0"/>
    <xs:element ref="GDE" minOccurs="0"/>
    <xs:element ref="GDE" minOccurs="0"/>
    <xs:element ref="GDE" minOccurs="0"/>
    <xs:element ref="GDE" minOccurs="0"/>
    <xs:element ref="GDE" minOccurs="0"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="PSE-Tbody">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="PSE-Row" maxOccurs="unbounded"/>
   </xs:sequence>
```

```
    </xs:complexType>
  </xs:element>
  <xs:element name="GDE">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="SDE" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="SDE" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="GDE" minOccurs="0"/>
        <xs:element ref="PSE-Tbody"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="PSE-Callout">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="GDE"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

L'élément `J2008`, racine de ce modèle, est un élément purement structurel. Il semble naturel qu'un élément racine soit de type `PSE`, cependant c'est loin d'être le cas avec toutes les DTD, mais aussi les schémas.

Sur les quatre éléments déclarés directement sous l'élément racine `J2008`, trois sont des éléments purement structurels, mais le premier (`ListofSIEs`) ne l'est pas.

La figure 9-12 illustre le premier niveau de l'arbre sous la racine. Elle montre clairement que les éléments `SIE` et `SIEdelete` sont deux éléments vides, pouvant être répétés de 1 à n fois. C'est pourquoi l'élément parent `ListofSIEs` n'est qu'un élément de données global.

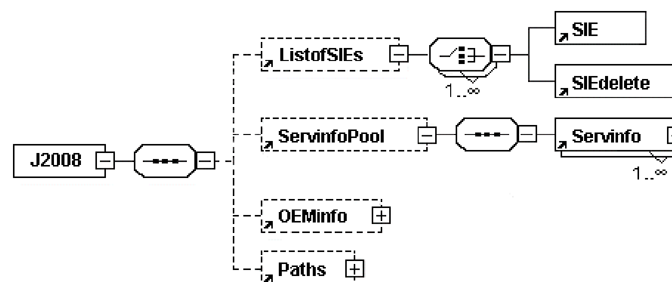


Figure 9–12
Premier niveau de l'arborescence

L'élément `OEMinfo`, qui est purement structurel, contient lui-même quatre sous-éléments purement structurels (`VehicleVars`, `ConfigVars`, `ConfigVarYrs`, `VehConfigVarYrs`). Quand on étend le sous-arbre sous `OEMinfo`, on se rend compte que les autres éléments sont pour la plupart des parents d'éléments vides. C'est la raison pour laquelle ils ne sont que du niveau `GDE`, comme le montre le graphe illustré à la figure 9-13.

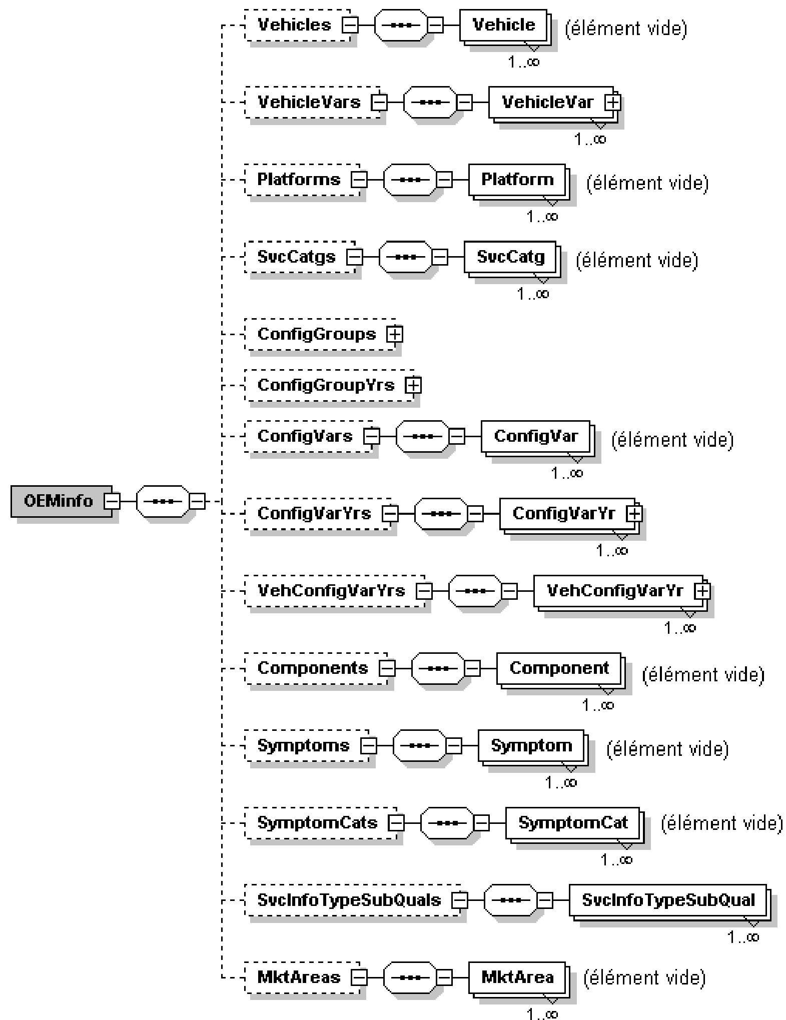


Figure 9–13
Structure de la branche OEMinfo

Dans la figure 9-14, nous allons nous intéresser à l'élément `table` dont on va voir qu'il a une structure particulière dans ce schéma. Cet élément représente un tableau, au sens documentaire, composé de rangées, de cellules et de texte dans les cellules. En général, les tableaux à destination de la documentation sont des objets dont le balisage est très structurant et l'élément `table` est un élément purement structurel. Or, ici, les tableaux, comme nous allons le voir, se dévoilent comme étant très irréguliers, utilisant en alternance tous les types d'éléments.

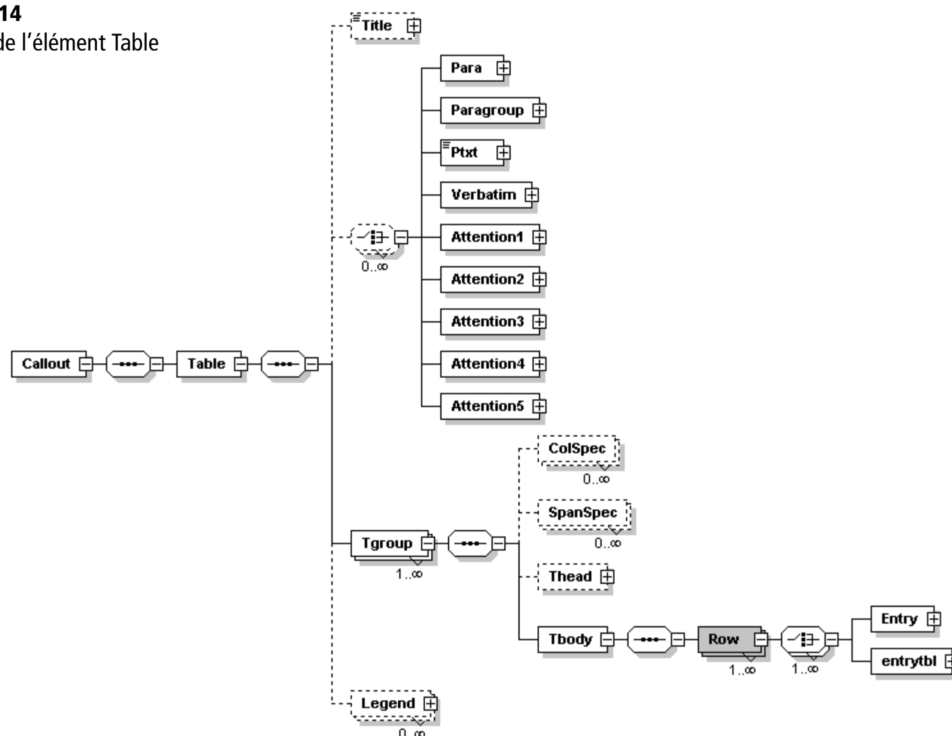


Figure 9–14
Structure de l'élément Table

Le seul élément purement structurel de haut niveau est l'élément `Callout`, parent de l'élément `table`. Ce dernier est perturbé par la présence de `Title`, un élément de type `SDE` placé directement sous `table`, et par la présence d'un choix important d'éléments de données (`para`, `paragroup`, `verbatim`, etc.). Il faut descendre jusqu'aux éléments `Tbody` et `Row` pour retrouver des éléments purement structurels.

Nous avons volontairement laissé le modèle dans cet état pour mettre en évidence ses caractéristiques. Dans la réalité, nous lui aurions appliqué la règle n°1 et l'aurions régularisé. Le seul élément purement structurel découvert au terme de l'analyse de cette sous-branche aurait été `Callout`, mais il aurait été lui-même absorbé par son parent

`Figure`, qui se révèle être de type `GDE !` De ce fait, la qualité d'élément purement structurel de l'élément `Callout` aurait été perdue et toute la branche des tableaux serait devenue indissociable du reste du document. Dans ce schéma, le tableau est solidaire du reste du document et ne peut être mis dans un fichier à part (dans l'intention par exemple d'en gérer le cycle de vie de manière différente du reste du document).

Par rapport à la méthode habituelle qui consiste à comprendre le rôle individuel de chaque élément, cette analyse par identification des éléments purement structurels met rapidement en évidence les singularités du schéma. Il est recommandé de commencer par s'attacher à comprendre les articulations macroscopiques d'un schéma avant de se lancer dans une analyse détaillée du rôle de chaque élément. Cela est particulièrement vrai quand on reçoit un schéma aussi volumineux que celui pris ici en exemple.

Pour en terminer avec les particularités du schéma que nous venons d'étudier, et notamment celle de l'élément `Callout`, dont la singularité est d'être un élément purement structurel absorbé, il est légitime de se demander à quoi sert cet élément qui ajoute un niveau de profondeur dans l'arborescence sans apporter aucune dimension structurelle supplémentaire (puisqu'il est absorbé). Un architecte d'application devra poser cette question aux concepteurs du schéma pour connaître la véritable raison de son existence. Il existe une bonne raison pour faire ainsi la chasse aux éléments qui ne servent à rien, car les coûts de mise en œuvre et de maintenance des schémas sont directement liés au nombre d'éléments qu'ils contiennent.

En résumé…

La méthode présentée dans ce chapitre permet :

- d'analyser un modèle de données XML sur le plan de sa structure intrinsèque ;
- d'avoir un regard synthétique sur la macrostructure d'un schéma XML ;
- de distinguer les éléments entre eux, et de faire la part entre ceux qui sont importants pour la gestion et ceux qui sont importants pour décrire les données ;
- de régler un schéma de manière qu'il reste souple dans différentes situations, et notamment qu'il puisse être adapté à différentes contraintes techniques ;
- d'écrire un modèle adapté à la cinématique d'un système d'information. Écrire un modèle n'est pas difficile dans le cadre d'une information statique, mais quand il s'agit de concevoir un modèle pour un système où l'information évolue, est échangée et transformée en permanence, mieux vaut prendre quelques précautions.

Après avoir étudié les possibilités de modularisation des modèles XML eux-mêmes, nous allons, au chapitre suivant, voir les particularités des documents XML qui sont des modules.

10

Concevoir des modules d'information

Nous allons maintenant aborder un concept important : celui des modules d'information.

Nous apporterons des réponses, illustrées par des exemples types, aux questions suivantes :

- Qu'est-ce qu'un module ?
- Comment définir la taille d'un module ?
- Quelles sont les règles de conception des modules ?

Les objectifs de la conception en modules d'information sont simples : il s'agit de stocker et gérer l'information sous la forme de granules afin de pouvoir reconstruire à volonté toute combinaison jugée utile de ces dernières. Comme il est préférable de ne pas dupliquer l'information, ce procédé demande un effort conséquent de mise en œuvre.

L'approche modulaire concerne tous les documents XML qui accompagnent des produits ayant plusieurs configurations, vendus dans plusieurs pays, avec des variantes par pays, voire par client. Ces documents sont tous fabriqués à partir d'une même base d'information mais diffèrent de par leur contenu ou leur mise en page. Il peut s'agir de notices techniques, catalogues, manuels d'utilisation, ou encore de contrats d'assurances ou de textes légaux.

L'idée de disposer d'un système d'information constitué d'unités faciles à réutiliser dans différents contextes et autonomes du point de vue de leur gestion est satisfaisante tant du point de vue fonctionnel qu'intellectuel. En effet, le fait de disposer d'un mécanisme pour combiner intelligemment, automatiquement et à la volée ces modules d'information en fonction de besoins immédiats renvoie à une vision du monde de l'information facile à appréhender par un esprit humain.

Concrètement cependant, qu'est-ce qu'un module ? Comment le définir ? Quelles sont les règles à respecter ? Telles sont les questions abordées dans ce chapitre.

Concernant le vocabulaire utilisé pour désigner un module, si ce terme s'impose peu à peu, on utilise encore les locutions homonymes *d'unité d'information*, *granule*, *information élémentaire*, et le terme de module lui-même se décline au travers des expressions suivantes : *module d'information*, *module de données* ou encore *module documentaire*.

Définir un module d'information

Dans le chapitre 3, nous avons vu comment découvrir les modèles logiques à partir de modèles conceptuels de données. Ce chapitre prolonge l'analyse des modules d'information en y intégrant les spécificités propres à XML.

« Ils savent tout et rien de plus » : cela pourrait être la définition d'un module d'information.

Plus prosaïquement, selon la définition officielle, un module est « une unité d'information adressable en tant qu'entité », ce qui signifie en clair :

- *Une unité* : un module d'information est un ensemble d'informations physiquement délimité, tel un fichier.
- *Une entité* : l'unité est reconnaissable par un identifiant unique.
- *Adresser* : l'entité est accessible par une adresse physique de type URL ou chemin d'accès (on dirait plus simplement : « on sait où la trouver »).

Comme unité d'information « adressable en tant qu'entité », un module d'informations se doit de satisfaire aux conditions propres intéressant les entités analysées et leurs règles de gestion, à savoir :

- être identifiable sans ambiguïté ;
- être stocké dans un fichier auquel on peut associer un identifiant unique de ressource, ou URI (Unique Ressource Identifier) ;
- exister temporellement (les changements d'états subis au cours d'un processus de production doivent pouvoir être parfaitement identifiés) ;

- être porteur d'une logique informative autosuffisante (le module doit contenir une information significative, non seulement pour les acteurs de son cycle de production, mais encore pour les applications utilisant les données qu'il contient) ;
- être plein et entier, c'est-à-dire être réutilisable sans modification de contenu dans plusieurs documents XML ; il s'agit d'un principe d'économie encore appelé *mutualisation des composants*.

La définition pratique d'un module se doit d'être cohérente avec les contraintes imposées par les processus (workflow) de création et diffusion du système d'information. Le modèle XML retenu doit être en adéquation avec les attentes de ce dernier.

Un module n'est pas un extrait quelconque du modèle de données ; quant à la définition de la taille des modules, ce n'est pas un simple problème de découpage en petits morceaux d'un schéma XML.

Créer des modules à la bonne taille

L'architecte des modèles XML vise principalement à obtenir le meilleur compromis possible entre les niveaux de granularité et l'efficacité fonctionnelle du système d'information. Nous présentons dans cette section une méthode permettant de créer des modules « de bonne taille ».

Il ne s'agit pas ici de définir une quelconque longueur physique ou un poids en octets, mais une dimension logique permettant de répondre à la question : « Sur quels nœuds de l'arbre commencent et se terminent les modules ? »

Contrairement à ce que l'on pourrait penser, il ne revient pas aux utilisateurs de définir la taille logique de leurs modules. Comme nous allons le voir, ce travail de conception revient à l'architecte des schémas XML. Dans un système d'information, les rôles des humains, des services et fonctions du système – voir l'étape 1 – et les modèles de données – voir les étapes 2 à 7 – sont liés. Comprendre et maîtriser cette interdépendance est véritablement la clé de voûte de tout succès dans la conception ; clé de voûte qui tient le pont entre modèles physiques de données et utilisateurs du système.

La méthode que nous mettons en avant rejoint les considérations des urbanistes des systèmes d'information. Il arrive un moment où les contraintes fonctionnelles doivent équilibrer celles pesant sur les données. On arrive alors à un point d'équilibre qui fait que, du moins en théorie, le système fonctionne harmonieusement d'un point de vue statique : il donne aux fonctions les données juste nécessaires.

Toutefois, une troisième force vient perturber ce bel équilibre : les processus. La méthode que nous allons décrire tout au long de ce chapitre a pour vocation de trouver dans les modèles le point d'équilibre entre données, fonctions et processus. En fournissant une démarche logique de découpage de l'information en modules

métier, et adaptant les schémas XML en conséquence, on répond à la préoccupation des urbanistes : trouver l'équilibre statique du système et l'étayer si solidement qu'il ne soit pas déstabilisé par les processus.

Pour créer des modules à la bonne taille, nous allons étudier la représentation graphique d'un schéma (voir la figure 10-2) et le parcourir de haut en bas, puis de bas en haut, en gardant à l'esprit la théorie des éléments purement structurels qui a été vue au chapitre 9. Le nom donné à la méthode renvoie à cette lecture verticale d'un schéma XML : son intitulé est en effet « méthode d'analyse *top-down* et *bottom-up* ». Son application permet de tracer une frontière entre la partie haute de l'arbre et la partie basse, et d'obtenir, dans la partie basse, des blocs verticaux parallèles, qui deviendront des modules. *In fine*, elle permet de superposer le modèle de données au modèle des rôles, ou acteurs d'un processus.

VOCABULAIRE Modèle des rôles

C'est selon le modèle des rôles qu'on attribue leurs rôles aux acteurs identifiés d'un workflow. Par exemple, un même programme peut extraire une donnée, la transformer, la valider et la remettre dans le système. Cet unique acteur a ici quatre rôles différents. Cet exemple fait bien comprendre l'intérêt d'avoir un modèle des rôles compatible avec le modèle de données.

VOCABULAIRE Workflow

Workflow est désormais un terme courant du jargon franglais de l'informatique, mais en français-français on doit dire processus. Un processus est une séquence de tâches, que l'on peut représenter formellement de manière graphique. Dans notre domaine technique, le mot processus désigne exclusivement la liste, les acteurs et l'ordre des étapes de changements d'états d'un objet. Qu'il s'agisse du mot workflow ou du mot processus, ils sont mis à contribution dans différentes expressions telles que processus qualité, processus industriel, processus métier, workflow de courrier, workflow des documents techniques… Le mot anglais en lui-même ne signifie pas grand-chose : le dictionnaire Harrap's le traduit par *rythme de travail* !
Un processus est une somme de changements d'états, lesquels sont provoqués par une action humaine ou automatisée, réalisée par un acteur qui a un rôle. Cet acteur peut être un programme ou un humain. Un même acteur peut bien sûr jouer plusieurs rôles à différents endroits d'un processus et un même objet peut subir plusieurs changements d'états. Une fois l'état d'un objet changé, une action de suivi peut être spécifiée.

Dans le schéma de la figure 10-1, on donne une représentation symbolique des liens entre processus et données d'assemblage. Chaque activité d'un processus utilise les fonctions mises à disposition par le système pour la manipulation des données. Nous avons symbolisé ces interactions entre données et processus par des flèches allant des étapes du processus aux données concernées. L'enchaînement des activités du processus montre l'imbrication des appels des fonctions d'édition ou d'assemblage des données. Lorsque les schémas ne sont pas conçus sous la forme de modules d'infor-

mation, eux-mêmes regroupés à l'aide d'éléments d'articulation, chaque action de modification touche l'ensemble de la structure arborescente du document. Cela conduit à des systèmes applicatifs de type spaghetti où l'imbrication entre les données et les traitements est telle qu'il est impossible de les faire évoluer.

La méthode décrite dans les paragraphes suivants permet, à partir d'une structure XML donnée, de faire apparaître les modules d'information et leurs éléments d'articulation.

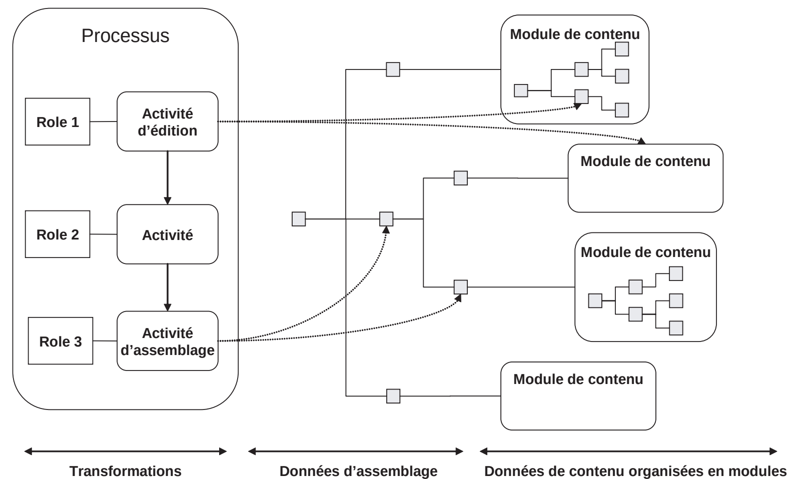


Figure 10–1
Schéma des trois interactions entre les étapes d'un processus et les ensembles de données

Pour illustrer la mise en œuvre de cette méthode, nous avons choisi la norme militaire américaine 38784c, qui fait partie de l'historique standard CALS (Computer aided Acquisition and Logistics Support) de 1988. Un schéma 38784c représente un document technique potentiellement volumineux (plusieurs milliers de pages) ; c'est la raison pour laquelle l'étude de son découpage logique est nécessaire. Nous avons choisi un cas issu du monde des documents « papier », considéré comme étant plus parlant qu'un cas extrait du monde des données. La démarche eût toutefois été la même avec un schéma XML orienté données.

La figure 10-2 illustre les premiers niveaux de ce schéma, très classiquement constitués d'éléments aux noms évocateurs : `doc`, `volume`, `front`, `body`, `rear`, `chapter`, etc. La structure du document est composée de trois éléments, `front`, `body` et `rear` (repère ❶), qui se répètent régulièrement à différents niveaux.

Les nœuds intermédiaires du schéma séparent l'élément racine `doc` des éléments `chapter` (repère ❷). Quand on expose le contenu de l'élément `chapter`, on obtient le graphe de la figure 10-3.

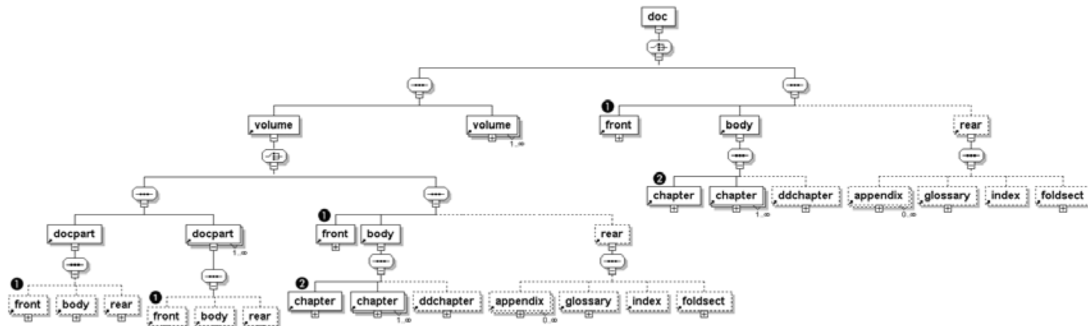


Figure 10–2 Les premiers niveaux de notre schéma exemple

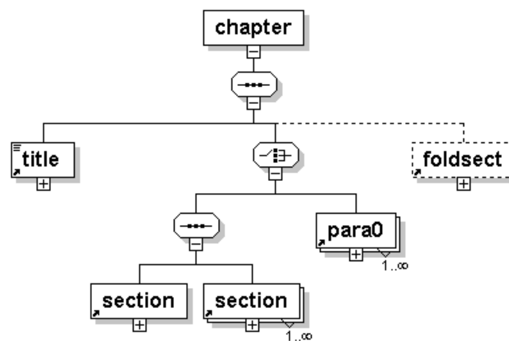


Figure 10–3
Le modèle de contenu de l'élément chapter

Cette branche fait apparaître les éléments `title` et `para0` qui ne peuvent pas être dissociés des autres : il s'agit d'éléments de données (voir le chapitre 9) et leur gestion séparée sous forme de modules n'aurait pas de sens en soi.

Quant aux éléments `section` qui se trouvent sous l'élément `chapter`, leur nom suffit à donner une idée de leur sémantique (voir figure 10-4). Une section est, de toute évidence, un type d'élément qui correspond fonctionnellement à l'idée qu'on peut se faire d'un module.

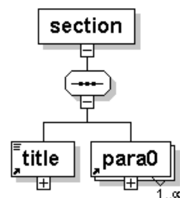


Figure 10–4
Le modèle de contenu de l'élément section

Cette analogie entre le nom de l'élément, sa fonction et un éventuel rôle de module est plus difficile à établir pour l'élément `foldsect`, dont le nom ne suffit pas pour se faire une idée de son type de contenu. Nous sommes donc obligés d'exposer son modèle de contenu, comme illustré à la figure 10-5. On obtient une structure combinant éléments textuels et appels de figures.

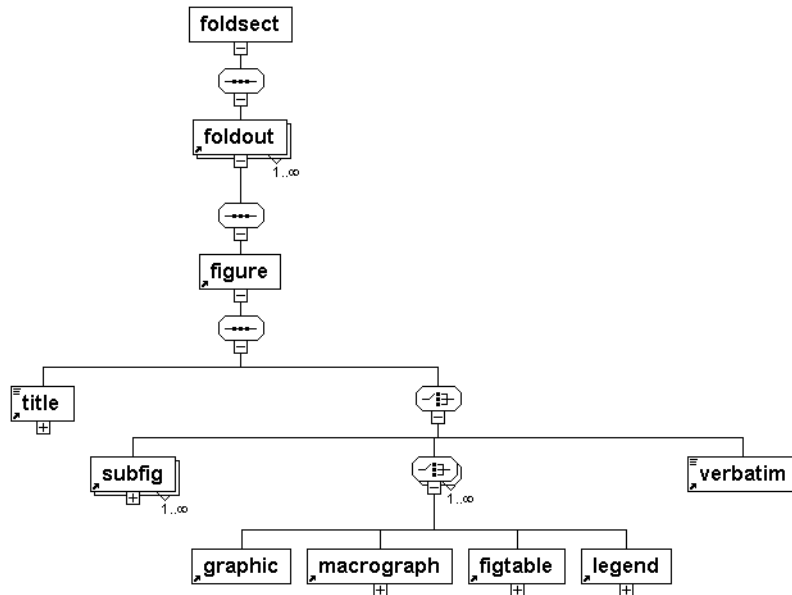


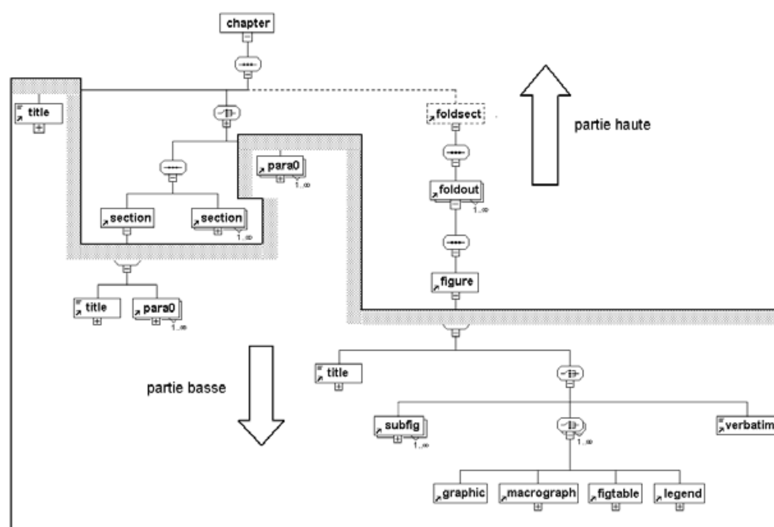
Figure 10–5
La branche de l'élément foldsect

Le modèle présenté ici est très irrégulier : certaines branches permettent d'accéder en très peu d'étapes à des éléments de type paragraphe tel `para` – nous assimilons à ce nom générique tous les éléments qui sont des conteneurs de texte –, alors que d'autres branches sont longues à dérouler, comme nous venons de le voir avec le cas de la branche de l'élément `foldsect`. Dans ce schéma, certaines branches nécessitent de parcourir une dizaine de nœuds avant d'arriver à un élément terminal de type textuel.

Comme expliqué précédemment, la méthode *top-down* et *bottom-up* permet de séparer cette grande structure en deux parties supérieure et inférieure. Dans chaque branche de l'arborescence, nous allons faire passer une frontière, juste au-dessus du premier élément de la branche de type textuel.

Appliquée à la structure de l'élément `chapter`, cette méthode donne le résultat illustré à la figure 10-6.

Figure 10–6
Séparation des parties haute et basse de la branche chapter



Le résultat est un arbre découpé en deux parties. La méthode est donc différente de celle utilisée pour les éléments purement structurels, cette dernière aboutissant quant à elle à un découpage en n parties.

Pour étendre la méthode à l'ensemble du schéma, depuis l'élément `doc` jusqu'aux premiers éléments de type `para`, nous allons simplifier le modèle et nous affranchir de tous les petits éléments textuels qu'il contient, de façon à obtenir une représentation graphique de taille raisonnable.

Pour cela, nous changeons le nom de tous les éléments terminaux de type textuel, que nous remplaçons par `para0`. Les branches contenant à la fois des `para0` et des sous-éléments de données globaux sont également remplacées par un seul élément `para0`. Cette méthode rappelle les règles de la théorie des éléments purement structurels expliquée au chapitre 9.

Le résultat illustré à la figure 10-7 met en évidence les deux parties fonctionnelles du modèle étudié. Dans la partie haute, se trouvent les éléments qui permettent d'assembler les modules. La partie basse regroupe les branches de l'arbre qui vont constituer les modules, et qui, pour l'instant, ont tous `para0` comme élément racine.

À ce stade de la présentation de la méthode, il nous paraît important de souligner l'importance de la frontière ainsi établie entre données et traitements, dans la continuité de ce que nous avons expliqué au chapitre 7 sur les différentes couches de programmation.

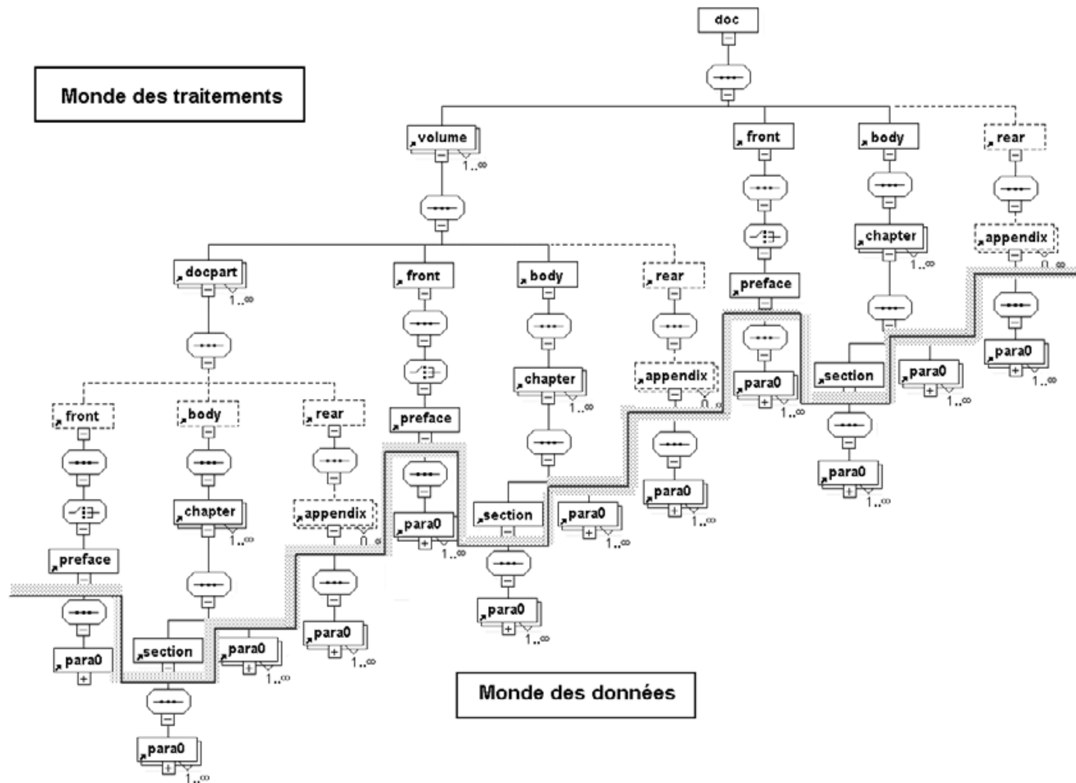


Figure 10–7 Frontière délimitant les parties haute et basse de notre schéma exemple

Les racines et le feuillage : les deux parties de l'arbre

Le découpage que nous venons d'effectuer va beaucoup plus loin que la seule identification de modules. Il met en évidence deux mondes différents et complémentaires :

- La partie haute représente le monde de l'assemblage et relève de la gestion de contenu, des processus et des traitements.
- La partie basse représente le monde de la fabrication du contenu et relève du travail des rédacteurs techniques, auteurs et programmes de production des données élémentaires.

La nature des systèmes informatiques à utiliser dans l'un et l'autre de ces deux mondes est très différente. Dans l'un, on voudra savoir comment la donnée est saisie, récupérée ou synthétisée ; c'est le monde de la gestion des données. Dans l'autre, on s'intéressera aux programmes d'assemblage, publication et diffusion avec comme terreau la gestion des documents au sens traditionnel de la GED (Gestion électronique de documents).

Vérifier la nature des éléments racines des modules

Les modules sont des entités qui doivent avoir une vie autonome. Ils ne sont rattachés à un « tout » plus important (en philosophie, on dit à un « englobant ») qu'au moment de la publication finale d'un document, d'une page web… Bref, d'un média quelconque.

C'est pourquoi la dernière étape de la méthode va consister à contrôler l'autonomie de ces modules. Ils doivent être aisés à décrocher du reste de l'arbre. En d'autres termes, l'élément parent de leur racine doit être un élément purement structurel. De cette manière, on contrôle que les résultats de l'analyse de la logique informative du schéma sont cohérents avec ceux de l'analyse des processus. L'élément purement structurel joue ce rôle de clé de voûte dont il était question dans l'introduction de ce chapitre, où il était affirmé que la méthode permettait de trouver le point d'équilibre entre données, fonctions et processus.

L'élément racine de chaque module doit être une articulation du schéma. C'est une règle de base qu'il faut respecter. En effet, si un module est un sous-document indépendant du document XML principal, il en contient potentiellement toutes les variations possibles autorisées par le schéma. Garantir que cet élément est purement structurel revient à garantir son rôle d'articulation dans tous les cas de figure.

Quand il apparaît que l'élément racine d'un module n'est pas un PSE, il faut choisir entre ajouter un élément parent à cette racine – cela suffit généralement à la transformer en PSE – ou élaguer les branches qui l'empêchent d'être un PSE. Le plus souvent, l'élément n'est pas un PSE car il est encadré sur sa gauche et/ou sa droite d'éléments textuels ; c'est le cas des éléments `chapter` et `title` de la figure 10-3.

Pour les éléments de gauche, il peut s'agir de métadonnées, ou données descriptives du reste de la branche (un titre, par exemple). Si tel est le cas, on tentera de modifier le modèle afin que ces données deviennent des attributs de la racine du module. Ainsi, un titre sous l'élément `chapter` pourrait devenir un attribut de cet élément. Les données de droite doivent être regroupées quant à elles sous un nouvel élément. Faute de satisfaire à ces modifications, la clé de voûte ne pourra réellement exister et le système sera toujours bancal.

Créer les articulations de la structure

Pour garantir la flexibilité des schémas, il faut y introduire, comme nous l'avons vu, des clés de voûte. Il s'agit d'éléments autour desquels s'articulent les différents sous-ensembles, sous-arbres et fragments dont on aura besoin dans les sous-systèmes et qui auront été définis au terme de la phase d'analyse des processus de production.

Nous allons voir de façon concrète dans cette section comment s'écrivent ces clés de voûte.

Comme toute articulation, elles sont en réalité composées de deux éléments qui s'emboîtent l'un dans l'autre : l'un est le fils de l'autre. L'ensemble se comporte comme un attelage : l'un joue le rôle d'anneau et l'autre de crochet, comme illustré à la figure 10-8.

Dans la partie droite de la figure, les éléments anneaux sont représentés en noir et les éléments crochets en gris. Les anneaux peuvent recevoir plusieurs éléments crochets, ce qui est le cas de l'élément n°3, ou d'autres éléments anneaux, comme l'élément n°2. La fonction de ces éléments est de faciliter le découplage des sous-arbres. Quand une telle séparation est opérée, les éléments anneaux deviennent des éléments terminaux, et les éléments crochets les racines des fragments détachés.

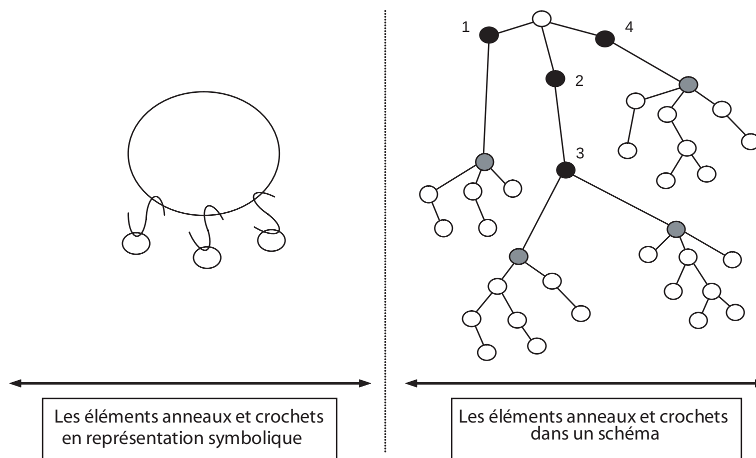


Figure 10–8
Les articulations, ou clés de voûte, d'un schéma XML

Considérons trois schémas, dont l'un représente la somme des deux autres. Ces schémas font intervenir deux éléments réellement appelés `anneau` et `crochet` pour les besoins de notre exemple. Nous allons montrer comment ces éléments doivent être définis en XML Schema pour que l'anneau soit un élément alternativement de type vide ou complexe, et le crochet alternativement racine d'un document XML valide ou enfant de l'anneau.

L'objectif final n'est évidemment pas de disposer de trois schémas mais de deux, complémentaires, qui, combinés par le jeu des inclusions, en formeront un troisième. Pour la clarté de notre exposé, ce troisième schéma sera appelé schéma global, tandis que les deux autres seront appelés schéma de l'anneau et schéma du crochet.

Voici le schéma de l'anneau :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="meta-structure">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="anneau" type="anneauType" nillable="true"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:complexType name="anneauType">
  <xs:simpleContent>
   <xs:extension base="xs:string"/>
  </xs:simpleContent>
</xs:complexType>
</xs:schema>
```

Remarquez que nous sommes obligés de définir l'élément `anneau` au travers d'un type global, en l'occurrence `anneauType`. Cela nous est imposé par les règles de XML Schema sur la redéfinition, laquelle ne s'applique qu'aux types, qui plus est globaux.

Le schéma du crochet :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="crochet">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="para" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 </xs:element>
</xs:schema>
```

Ce schéma n'a rien de spécifique.

Le schéma global :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:include schemaLocation="crochet.xsd"/>
 <xs:redefine schemaLocation="anneau.xsd">
  <xs:complexType name="anneauType">
```

```
    <xs:complexContent>
     <xs:extension base="anneauType">
      <xs:sequence>
       <xs:element ref="crochet" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>
  </xs:redefine>
</xs:schema>
```

Ce schéma contient une inclusion simple, celle du schéma du crochet, et une redéfinition, celle du schéma de l'anneau. Le schéma global est donc limité aux seules définitions de ses propres éléments et aux redéfinitions des éléments de jointure.

Nous présentons ci-dessous les documents XML correspondants.

Document XML de l'anneau :

```
<?xml version="1.0" encoding="UTF-8"?>
<meta-structure xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation="anneau.xsd">
 <anneau/>
</meta-structure>
```

Ce document ne contient qu'un seul élément vide `anneau` et son parent `meta-structure`.

Document XML du crochet :

```
<?xml version="1.0" encoding="UTF-8"?>
<crochet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="crochet.xsd">
 <para>blabla</para>
</crochet>
```

Dans ce document, l'élément `crochet` est racine.

Document XML de l'ensemble :

```
<?xml version="1.0" encoding="UTF-8"?>
<meta-structure xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation="ensemble.xsd">
 <anneau>
  <crochet>
   <para>blabla</para>
```

```
  </crochet>
 </anneau>
</meta-structure>
```

Dans une application réelle, l'idéal serait d'utiliser le mécanisme d'inclusion de XInclude, comme ci-après :

```
<?xml version="1.0" encoding="UTF-8"?>
<meta-structure xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="chap08-anneau-crochet.xsd">
 <anneau>
  <xi:include href="crochet.xml" parse="xml" encoding="UTF-8"
             xmlns:xi="http://www.w3.org/2001/XInclude/">
 </anneau>
</meta-structure>
```

Dans cette section, nous avons rappelé l'importance qu'il y a à tenir compte des processus de création, gestion et publication des documents XML pour découvrir leurs éléments de structuration et de modularisation. La méthode d'analyse qui a été présentée permet d'adapter le schéma initial (en général issu de la seule analyse du modèle conceptuel des données) et d'y introduire des éléments `anneau` et `crochet` qui servent de clés de voûte des systèmes bâtis sur le concept de modules.

Règles de conception applicables aux modules

Une fois les modules identifiés et leurs éléments racines établis, quatre règles de pure logique s'appliquent :

- **Règle de non-emboîtement.** Les modules ne doivent pas être emboîtés les uns dans les autres, mais uniquement juxtaposés.
- **Règle d'applicabilité.** Un module ne peut en aucun cas contenir dans son texte une information qui serait de nature à modifier ses règles d'assemblage avec d'autres modules.
- **Règle de balisage.** Les modules traduisent la volonté de réutiliser une même information à plusieurs endroits ; leur balisage doit donc être souple, flexible, facilement réutilisable : on dit que le balisage doit être polymorphe.
- **Règle d'identification des éléments.** La puissance d'un système modulaire tient dans les liens qui vont pouvoir être tissés entre les modules : ces liens doivent s'adapter naturellement à toute configuration d'assemblage des modules.

Nous expliquons ces règles dans les prochaines sections. Toutefois, il ne s'agit là que de recommandations. Libre à vous de ne pas les suivre. Vous trouverez alors peut-être que votre système fonctionne ; il sera simplement plus coûteux à mettre au point et plus difficile à maintenir.

Règle de non-emboîtement

Cette règle d'apparence simple est difficile à identifier quand on ne la connaît pas, et la tentation est grande d'autoriser les modules à s'emboîter les uns dans les autres. C'est l'erreur la plus fréquemment rencontrée et la plus grave. Nous allons l'expliquer au travers d'un exemple concret.

Nous reprenons à la figure 10-9 la représentation graphique de l'élément `chapter`. La présence du sous-élément `section` donne envie de découper la branche `chapter` de sorte que les éléments `section` soient des modules (repères ❶ et ❷). Les éléments `figure`, chapeautés par l'élément purement structurel `foldout`, sont de bons candidats pour faire des modules (repère ❹). Par analogie entre `section` et `foldsect`, on peut envisager de faire un module de l'élément `foldout` (repère ❸). Enfin, pour éviter le problème de l'élément `title` sous l'élément `chapter`, on peut décider que l'élément `chapter` lui-même est la racine du module (repère ❺). Sur la figure, les encadrés en pointillé représentent les modules possibles de l'élément `chapter`.

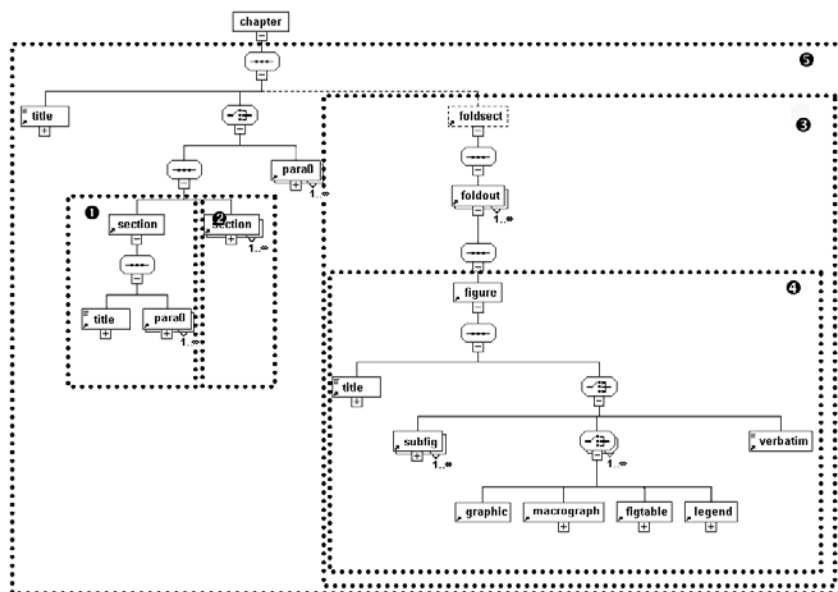


Figure 10–9
Mise en évidence des modules envisagés sur la représentation graphique d'un schéma

Sur la vue balisée d'un document XML, l'organisation en modules de la figure 10-9 peut être représentée par la figure 10-10 :

Figure 10–10
Mise en évidence des modules envisagés sur la représentation balisée d'un document XML

```
<chapter>
 <title>Ceci est le titre du chapitre</title>
 <section>
  <title>Ceci est le titre de la section</title>
  <para0>Ceci est un paragraphe</para0>
 </section>
 <section>
  <title>Ceci est le titre de la section</title>
  <para0>Ceci est un paragraphe</para0>
 </section>
 <foldsect>
  <foldout>
   <figure>... ...</figure>
  </foldout>
  <foldout>
   <figure>... ...</figure>
  </foldout>
  <foldout>
   <figure>... ...</figure>
  </foldout>
 </foldsect>
</chapter>
```

Or, un module ne peut en contenir d'autres ; l'emboîtement est interdit. Il faut choisir entre :

- conserver le module du niveau `chapter` et supprimer tous les autres,
- supprimer le module de niveau `chapter` et conserver les modules de niveau `section`,
- supprimer le module de niveau `foldsect` et ne conserver que les modules de niveau `figure`,
- conserver le niveau `foldsect` et supprimer les modules de niveau `figure`,
- supprimer les modules de niveau `foldsect` et `figure`, et créer des modules de niveau `foldout`.

Le découpage serait alors tel que le présente la figure 10-11.

La règle de non-emboîtement traduit la volonté d'éviter l'utilisation de liens d'inclusion entre les modules. Elle pourrait, de ce fait, être appelée *règle de linéarisation des appels de modules*. Elle met en évidence la structure qui formera le squelette d'une publication, que nous appelons dans cet ouvrage *backbone*, *méta-structure* ou *structure d'assemblage*. Dans notre exemple, les balises composant ce squelette sont `chapter` et `foldsect`.

Figure 10–11
Découpage finalement retenu après vérification de la règle de non-emboîtement

```
<chapter>
 <title>Ceci est le titre du chapitre</title>
 <section>
  <title>Ceci est le titre de la section</title>
  <para0>Ceci est un paragraphe</para0>
 </section>
 <section>
  <title>Ceci est le titre de la section</title>
  <para0>Ceci est un paragraphe</para0>
 </section>
 <foldsect>
  <foldout>
   <figure>... ...</figure>
  </foldout>
  <foldout>
   <figure>... ...</figure>
  </foldout>
  <foldout>
   <figure>... ...</figure>
  </foldout>
 </foldsect>
</chapter>
```

D'autres effets de bord sont évités par la règle de non-emboîtement : l'indépendance des modules facilite leur mise à jour en assurant une gestion des révisions avec des interférences limitées entre modules.

REMARQUE Élément ou attribut ?

Dans notre exemple, l'élément `titre`, sous `chapter`, empêche à lui tout seul l'élément `chapter` d'être purement structurel. En l'état du schéma de l'exemple, nous n'aurions donc pas pu déclarer `chapter` comme étant la racine d'un module. Nous avons préféré ne pas tenir compte de cette considération technique qui, en alourdissant l'exemple, aurait nuit à son côté didactique. Ce choix est admissible car, de type simple, l'élément `titre` peut ici être assimilé à un attribut de l'élément `chapter`. Ce faisant, `chapter` redeviendrait un élément purement structurel.

Règle d'applicabilité

L'applicabilité traduit une relation de dépendance entre une donnée et le système physique auquel elle s'applique. Ainsi, l'équipement d'une voiture s'applique à une partie seulement des véhicules d'une gamme. Le plus souvent, l'applicabilité est le miroir de la gestion de configuration des matériels documentés. Elle peut toutefois traduire d'autres critères tels que la langue, les particularités propres à un pays, un contrat, une compétence, voire un âge… Récemment, des critères de visibilité sont

venus s'ajouter aux critères d'applicabilité : la visibilité concerne les conditions selon lesquelles une information peut être consultée et imprimée.

Les termes applicabilité, configuration et révision ne doivent pas être confondus. La gestion des révisions consiste à gérer les modifications de contenu successives appliquées à un module au cours de sa vie (y compris pendant les différentes étapes de son processus de production, ou *workflow*), tandis que la gestion de configuration concerne l'ensemble des modules de données réunis à un instant et des circonstances données pour constituer une information de niveau supérieur.

Des exemples classiques d'applicabilité sont :

- la version du système mécanique ou logiciel documenté ;
- le client à qui le système est destiné ;
- les créneaux calendaires, du type « *du 23.05.1999 au 15.08.1999 et du 20.09.1999 au 13.12.2000* » ;
- les fuseaux horaires ;
- les zones géographiques et territoires commerciaux ;
- les habilitations (« seules les personnes autorisées peuvent lire ce paragraphe ») ;
- les langues du pays et du système, la langue utilisée dans un système n'étant pas obligatoirement celle de sa documentation ;
- la localisation et les habitudes culturelles.

Le plus souvent, les applicabilités influencent le texte qui est contenu dans un document, par exemple : *« Dans le cas des moteurs Benson, le volant moteur doit être calé par rapport au repère de gauche, tandis que dans le cas des moteurs Logosse c'est celui du milieu qui doit être utilisé. »* La portée d'une applicabilité peut être un mot, une donnée, une phrase, un paragraphe, une section, un module. Ici, nous donnons même un exemple pour montrer que le style rédactionnel lui-même peut être touché par une éventuelle gestion des applicabilités.

En effet, un des objectifs de la gestion des applicabilités est, pour des raisons évidentes de sécurité, d'occulter la partie du document XML qui ne concerne pas un lecteur dans une situation donnée. Dans notre exemple, un mécanicien ayant déclaré travailler sur du matériel Logosse devrait lire à l'écran : *« Pour caler le volant moteur, utilisez le repère du milieu. »*

Pour obtenir ce résultat, l'applicabilité est codée dans le document XML :

- soit au niveau des mots, comme ci-après :

```
<p>Pour caler le volant moteur, utilisez le repère <applic
case="Benson">de gauche</applic><applic case="Logosse">du milieu</
applic>.<p>
```

- soit au niveau des paragraphes, comme ceci :

```
<p applic="Benson">Pour caler le volant moteur, utilisez le repère de
gauche.<p>
<p applic="Logosse">Pour caler le volant moteur, utilisez le repère du
milieu.<p>
```

Les concepteurs des modèles de données ont la responsabilité de définir la stratégie de balisage (utiliser des éléments ou des attributs), de spécifier les éléments porteurs des informations d'applicabilité, ainsi que d'établir la syntaxe d'écriture des critères d'applicabilité. En effet, si le cas cité en exemple est simple puisqu'il n'existe qu'un critère d'applicabilité, que dire de ceux qui combinent configuration d'un système, numéro de version, date et version linguistique…

La multiplication des critères d'applicabilité montre rapidement qu'une telle information ne peut rester inconnue du système de gestion. La visibilité humaine et informatique de cette information est une condition *sine qua non* du bon fonctionnement de la gestion de contenu d'un système d'information. Les critères d'applicabilité doivent être exposés.

La règle d'applicabilité proclame qu'à l'intérieur d'un même module ne sont autorisées que des variations d'applicabilité d'une même famille, laquelle doit être visible à l'extérieur du module. La définition d'une famille d'applicabilités dépend du secteur industriel concerné. Dans notre exemple, la famille pourrait être définie ainsi : `toutes motorisation, moteurs=(Benson, Logosse)`, ou encore `107 Diesel tous pays toutes versions`…Tout dépend des habitudes, contraintes et cas réels de la profession. Du seul point de vue conceptuel, l'important est de ramener le problème à un jeu d'étiquettes non entrelacées : la famille d'applicabilités doit pouvoir être identifiée par un nom à partir duquel toutes les variantes d'applicabilités locales doivent pouvoir être déduites. Par exemple, il serait interdit de déclarer qu'un module de données s'applique aux « 107 Diesel tous pays toutes variantes », et de mentionner à l'intérieur une exception telle que *« sauf version 4B équipée des moteurs Atlas de 1963 »*.

L'applicabilité générale d'un module de données doit pouvoir être exposée simplement au niveau des métadonnées de gestion ; elles sont au module ce qu'une API est à un programme. Il est possible de déclarer sous forme de jeux d'étiquettes simples que le module s'applique globalement aux modèles de la famille des « 107 Diesel tous pays toutes variantes », mais il est impossible de déclarer qu'il s'applique aux modèles « 107 Diesel tous pays toutes variantes à l'exception du troisième paragraphe, qui ne s'applique pas aux versions 4B équipées des moteurs Atlas de 1963 ». Sans une telle règle, on n'en finirait pas d'énumérer toutes les possibilités et combinaisons d'applicabilités, sans parler de la gestion des révisions, qui deviendrait alors un problème sans solution.

Cette règle peut, à elle seule, provoquer une refonte en profondeur du modèle de données, de la traduction en XML de la logique métier et des interfaces d'accès du système d'information.

Règle de balisage

Quand on étudie le balisage d'un module, on est tiraillé entre deux contraintes antagonistes :

- utiliser un balisage porteur d'une sémantique précise ;
- assurer la flexibilité du système en augmentant les possibilités d'assemblage des modules. Pour cela, il faut que le balisage ne provoque aucune collision de validation au moment de l'assemblage des modules.

Le balisage doit donc être à la fois assez précis, puisque par définition proche de la donnée et donc des applications, et souple pour que tout assemblage d'un module avec un autre ne soit pas rejeté par le validateur. Ce dernier principe va très loin si l'on estime que n'importe quel module doit pouvoir être associé, sans contrainte, avec n'importe quel autre… y compris si les schémas internes et d'assemblage sont différents ! Les principes étudiés au chapitre 5 pour arbitrer entre éléments ou attributs pour le passage de la sémantique trouvent toute leur application dans la mise en place des modules d'information.

L'approche modulaire n'a de sens que si cette flexibilité est respectée. Il serait réducteur de disposer de modules qui ne pourraient être assemblés qu'avec leurs semblables. La valeur ajoutée d'un système de gestion de contenu orienté modules est précisément de pouvoir disposer d'une véritable base d'informations réutilisables à volonté sous différentes formes.

Dans l'approche modulaire, on ne peut savoir à l'avance et de façon sûre quelle place occupera un module dans une publication, tantôt section, simple paragraphe ou encore commentaire en aparté… Il n'est donc ni souhaitable ni facile de définir un balisage trop directif par rapport à l'usage final de l'information qu'il décrit. Au contraire, un balisage polymorphe est tout indiqué, qui puisse être adapté aux environnements dans lesquels les applications vont le plonger.

Ces considérations nous conduisent à affirmer qu'un module ne doit être composé que d'éléments parmi les plus banals qui soient (paragraphes, listes, titres, etc.) et que toute son intelligence doit être stockée dans des attributs. On est alors proche du balisage HTML, complété d'un balisage secondaire réduit à quelques balises permettant de spécifier librement la sémantique des données.

Dans le domaine des documents XML orientés données, on peut aussi obtenir des balisages neutres, ou banals, en utilisant des jeux de balises représentant principale-

ment les associations entre données plutôt que les données elles-mêmes. C'est ce que nous avons fait en reprenant l'exemple du chapitre 2, en figure 2-1, cette fois-ci en indiquant que la hiérarchie représentée graphiquement pouvait se traduire en XML soit par un balisage spécifique de l'application, soit par un balisage neutre et générique inspiré de Topic Map. Nous en donnons un exemple concret dans le tableau 10-1.

Tableau 10–1 Balisage spécifique vs neutre dans des documents XML orientés données

Balisage spécifique à une application	Balisage neutre inspiré de Topic Map
<artiste nom="Jean"> <troupe nom="Troupe du theatre de Lyon"/> <film_joue nom="grand bleu"/> </artiste>	<data classe="artiste" id="p1">Jean </data> <data classe="troupe" id="t1"> Troupe du théatre de Lyon </data> <data classe="film" id="f1">Le grand Bleu</data> <link> <source idref="p1" role="acteur"/> <target idref="f1" role="film"/> <arcrole>est acteur du film</arcrole> </link> <link> <source idref="p1" role="acteur"/> <target idref="t1" role="troupe"/> <arcrole>est membre de</arcrole> </link>

Avec l'approche modulaire, la question n'est plus de savoir si l'information contenue dans le module apparaît au début, à la fin, en haut ou en bas de l'arbre, mais de savoir quel sera le sens de parcours du lecteur avant d'arriver à l'information. On n'est plus dans un concept hiérarchique d'accès à l'information, mais dans une organisation neuronale basée sur des liens. On passe ainsi d'une organisation hiérarchique à une organisation totalement en réseau.

Le principe du balisage polymorphe traduit la volonté de rendre polymorphes les modules eux-mêmes. Cette règle s'applique également aux documents XML orientés données : il faut réfléchir à la porté sémantique du balisage qu'ils utilisent ; sans cet effort de réflexion sur la sémantique du balisage, ce ne serait qu'une pâle copie des tables relationnelles qu'ils interfacent.

Règle de limite inférieure de modularisation

Il se peut que vous soyez tenté de créer des modules de très petite taille sous prétexte que telles ou telles données (ou phrases) se trouvent réutilisées à l'identique à différents endroits d'un document XML. Or, contrairement à ce que l'on pourrait penser, il ne faut pas le faire, en tout cas pas sous la forme de modules.

Par exemple, la reprise à l'identique d'une même phrase à différents endroits d'un document n'a jamais signifié que cette phrase devait faire l'objet d'un module. La raison en est simple : sortie de son contexte d'utilisation, cette phrase n'a probablement aucun sens ; ce n'est pas une unité logique d'information. Ce qui se comprend simplement avec le cas d'une phrase se comprend encore plus avec une donnée élémentaire, telle que `24°5` : sorti de son contexte d'utilisation, ce chiffre ne veut rien dire.

Ainsi, il y a rupture totale entre les concepts de données élémentaires et celui d'unité d'information ou module. Le processus même de création d'une donnée élémentaire n'a absolument rien à voir avec celui de l'unité d'information qui l'utilise (on devrait dire qui l'exploite). On comprend aisément qu'il serait stupide de créer dix modules pour les chiffres de 0 à 9 sous prétexte qu'ils sont utilisés partout ! Faire cela serait évidemment absurde. De même que tout n'est pas objet, tout n'est pas décomposable à l'infini.

En clair, il existe réellement une limite inférieure à la décomposition des unités d'information. La mutualisation des composants des documents XML ne peut aller au-delà d'une certaine limite au risque d'aboutir à un phénomène de surpopulation qui provoquerait l'effet inverse à celui recherché : une paralysie complète du système d'information.

Cela ne veut pas pour autant dire qu'il n'est pas possible d'établir des listes ou catalogues de granules d'information élémentaires : des phrases types, des icônes types, des ensembles de phrases tels que des formules légales standard, sont par exemple de très bons candidats à une mise en bibliothèque d'objets types. Les bibliothèques peuvent être organisées par thèmes et suivre leur propre cycle de révision/mise à jour. Les objets qui s'y trouvent peuvent être référencés dans les documents par des expressions XPath, XPointer, ou de simples identifiants. En tout état de cause, ils ne font pas l'objet de modules indépendants ; c'est tout au plus des atomes d'information et seules les bibliothèques peuvent prétendre être considérées comme étant des modules.

Règle d'identification des éléments

Dans la même veine que le polymorphisme du balisage, les identifiants de références croisées doivent s'adapter en toutes circonstances au résultat de l'assemblage de modules, et ce, sans limitation liée à des contingences physiques.

Quand les liens inter-modules sont écrits en dur (comme la plupart des liens écrits dans les pages HTML), le système est rapidement gelé. C'est particulièrement vrai dans l'approche modulaire puisqu'un lien crée une relation de dépendance entre modules. Cela est contradictoire avec la volonté d'assembler des modules indépendants au gré des besoins. Si la seule utilisation d'un module obligeait à rapatrier ceux qui lui sont liés, c'est de proche en proche potentiellement toute la base qu'il faudrait rapatrier, à la manière d'une pelote de laine dont on commence à tirer le fil.

Pour préserver la souplesse d'un système modulaire, les modules ne doivent contenir que des liens logiques dont le nombre est réduit autant que faire se peut : une bonne modélisation conduit toujours à créer automatiquement un grand nombre de liens.

Dans les modules, les sources et cibles de liens doivent être définis logiquement. La valeur concrète d'un lien sera calculée au dernier moment, en fonction du média de publication.

Nous allons expliquer comment s'affranchir dans les modules des liens dits *point à point*, équivalents XML des URL de HTML. La technique que nous proposons est d'ailleurs inspirée des URL et repose sur la définition d'un couple composé de l'identifiant du module cible et de l'identifiant de l'élément ciblé à l'intérieur du module cible. Le tableau 10-2 en donne un exemple concret. Cette technique permet ultérieurement de passer à un niveau plus abstrait de liens.

Tableau 10–2 Exemple de lien en environnement modulaire

Module source	**Commentaire**
Le principe de l'identification consiste à considérer que tous les identifiants des éléments d'un module font partie d'une famille dont le nom est l'identifiant du module. En voici un exemple : <ref moduleCode="m456-786" target-type="table" target-id="t2" ref-type="number">.	Dans ce fragment, on spécifie *via* la balise ref une référence à un tableau se trouvant dans un autre module, à savoir celui dont l'identifiant est m456-786.
Module cible	**Commentaire**
<module id="m456-786" rev-number="2.5" applic="R4U,R4Z"> … <table id="t2"> <title> Exemple typique de référencement d'élément en environnement modulaire </title>. </title> … </table> </module>	Ce fragment est la cible de la référence ci-dessus. L'identifiant cible porté par l'élément table est très simple, la valeur t2 est indépendante de l'identifiant id du module, de son applicabilité applic et de son indice de révision rev-number. Ce module est contenu dans un fichier dont le nom peut être indépendant de son id, par exemple 456786.xml.

Au moment de l'assemblage des modules, les identifiants sont rendus uniques en accolant l'identifiant du module à celui de l'élément ciblé. La transformation en lien définitif est ensuite très simple. Elle suit la méthode présentée au tableau 10-3 et peut être résumée ainsi :

1. Première transformation : création d'identifiants uniques.
2. Deuxième transformation : déplacement des identifiants uniques nouvellement créés vers les éléments qui seront les vraies cibles physiques (par exemple, si un lien pointe logiquement vers une section d'un document, son lien physique correspondant pointera fort probablement vers le seul titre de ladite section).
3. Troisième et dernière transformation : fabrication des liens physiques dont la forme dépend du média de restitution de l'information.

Tableau 10–3 La chaîne de transformation des identifiants de références croisées

Étape 1. Création des identifiants uniques

Des identifiants uniques sont calculés. Nos modules initiaux deviennent :

```
<p>Le principe de l'identification consiste à considérer que tous les
identifiants des éléments d'un module font partie d'une famille dont le nom
est l'identifiant du module. En voici un exemple :<ref refid="m456-786-t2"
target-type="table" ref-type="number">.</p>
<table id="m456-786-t2">
<title>
Exemple typique du référencement d'un élément en environnement modulaire
</title>
...
</table>
```

Étape 2. Déplacement des identifiants vers les éléments porteurs

Déplacement des identifiants vers les éléments visibles après composition. Dans notre exemple, c'est le titre du tableau qui sera visible, pas l'élément `table`.

```
<p>Le principe de l'identification consiste à considérer que tous les
identifiants des éléments d'un module font partie d'une famille dont le nom
est celui du module. En voici un exemple : <ref refid="m456-786-t2"
target-type="table" ref-type="number">.</p>
<table>
<title id="m456-786-t2">
Exemple typique du référencement d'un élément en environnement modulaire
</title>
...
</table>
```

Tableau 10–3 La chaîne de transformation des identifiants de références croisées

Étape 3. Transformation physique en fonction du média de sortie
La dernière étape est le calcul de l'URL ou du renvoi textuel. **1er exemple** : production d'une URL ` Le principe de l'identification consiste à considérer que tous les identifiants des éléments d'un module font partie d'une famille dont le nom est celui du module. En voici un exemple : **au tableau « Exemple typique du référencement d'un élément en environnement modulaire »**. `
2e exemple : production d'une référence croisée pour une impression. Dans ce cas, on ajoute le texte qui apparaîtra dans la version imprimée Le principe de l'identification consiste à considérer que tous les identifiants des éléments d'un module font partie d'une famille dont le nom est celui du module. En voici un exemple : **au tableau** <ref refid="m456-786-t2" target-format="table-number"> **page** <ref refid="m456-786-t2" target-format="page-number">. ... <table> <title id="m456-786-t2"> Exemple typique du référencement d'un élément en environnement modulaire </title> ... </table>

La règle d'identification des éléments exige que les identifiants d'éléments soient indépendants des publications, des indices de révision et numéro de version des modules, ainsi que des opérations de copier-coller des éléments entre les modules ou au sein d'un même module.

Les structures d'assemblage

Dans l'approche *top-down* et *bottom-up*, la structure d'assemblage est un document XML qui traduit la partie haute de l'arbre.

La figure 10-12 en est un exemple facile à comprendre : on retrouve ici la table des matières logique d'un manuel de maintenance, que l'on présente juste après sous sa forme de squelette d'assemblage XML.

```
<manual system="JLPruvot" type="OM" idcode="2169329-100" revision="C"
subtitle="Preliminary version – for external evaluation prototype
only">
```

```
<heading title="Introduction">
  <heading title="Presentation of the system">
   <dm>&M1300009;</dm>
   <dm>&M1300010;</dm>
  </heading>
  <heading title="Composition">
   <dm>&M1300011;</dm>
   <dm>&M1300013;</dm>
  </heading>
  <heading title="Description of the components">
   <dm>&M1300015;</dm>
  </heading>
</heading>
 <heading title="Utilisation">
  <heading title="Examination preparation">
   <dm>&M6000017;</dm>
   <dm>&M6000018;</dm>
   <dm>&M6000019;</dm>
  </heading>
</heading>
 <heading title="Maintenance">
  <heading title="Planned maintenance">
   <dm title="Planned maintenance user schedule">&M3200025;</dm>
   <dm title="Planned maintenance FE's schedule">&M3200026;</dm>
  </heading>
  <heading title="User's procedure">
   <dm title="Planned maintenance user procedure start">&M3300027;</dm>
   <dm title="Planned maintenance user procedure end">&M3200028;</dm>
  </heading>
 </heading>
 <heading title="Message description">
  <dm title="User's error list">&M3700029;</dm>
 </heading>
</manual>
```

L'élément récursif `heading` définit le positionnement hiérarchique des modules dans la publication. Il traduit les imbrications de volumes, chapitres, sections…, qui composent toute publication, même s'il s'agit d'un catalogue ou d'un listing de données. La fonction de l'attribut `title` est double :

- transmettre un titre au système de gestion,
- qualifier un niveau afin de déclencher des événements particuliers de composition.

Dans l'exemple présenté, nous n'avons fait porter à l'élément `heading`, pour des raisons évidentes de simplification, aucune métadonnée autre que `title`. On pourrait

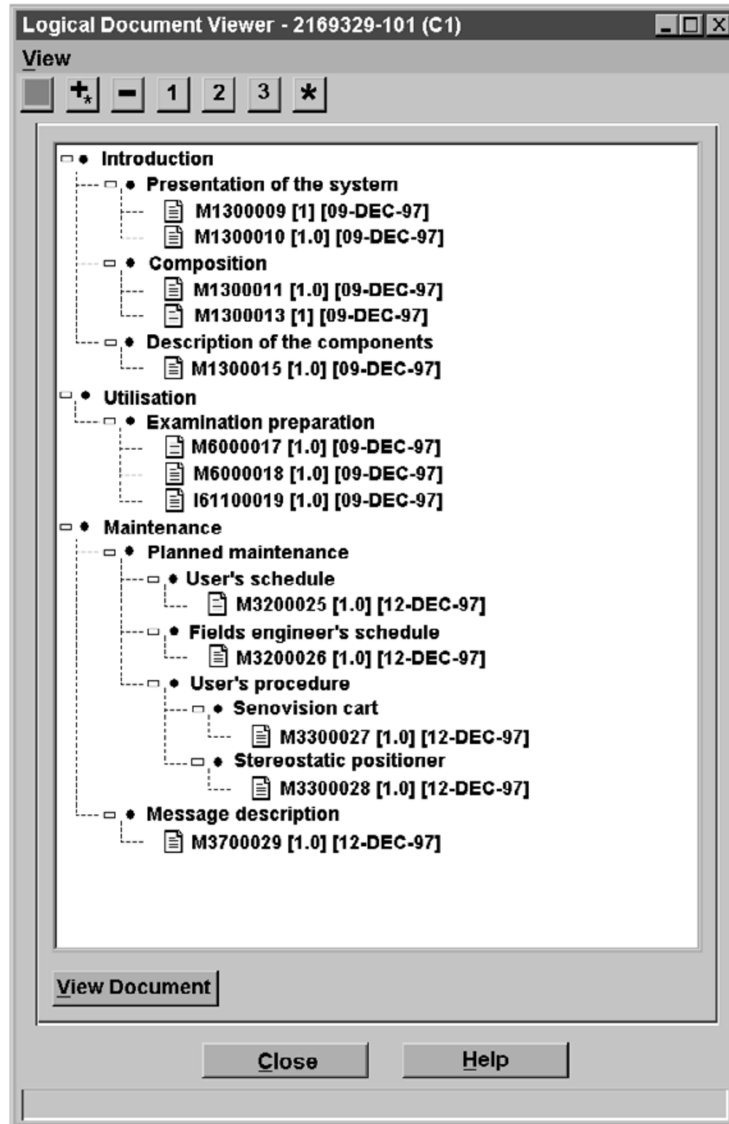


Figure 10–12
Exemple de structure d'assemblage

utiliser tout attribut utile pour transmettre aux applications autant d'informations que nécessaire. Quand le modèle modulaire est bien conçu, les attributs sont suffisants pour transmette aux applications la sémantique qui les concerne.

> **REMARQUE Structure d'assemblage**
>
> La figure 10-12 montre la version XML d'une structure d'assemblage. En règle générale, c'est le système de gestion qui gère, sous la forme qui lui convient, la structure d'assemblage.

La figure 10-13 montre la représentation graphique du schéma XML utilisé par notre structure d'assemblage, et la figure 10-14 en fournit la représentation UML. La particularité principale du modèle est la récursivité de l'élément `heading` : le modèle étant valable pour une grande diversité de cas de figure, le nombre de niveaux d'imbrication de cet élément n'est pas figé. C'est la couche métier qui donnera, lors des traitements finaux, son véritable sens, ou sémantique, à l'élément `heading` (pour respecter la règle de balisage décrite plus haut).

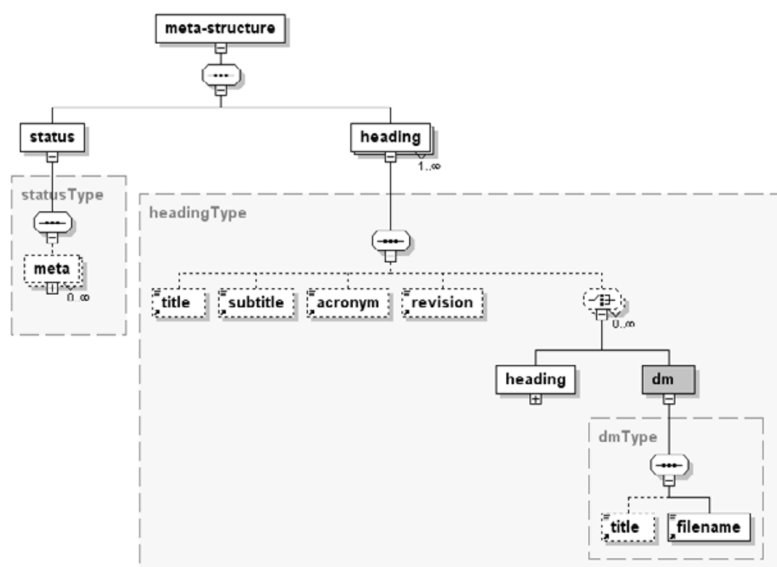


Figure 10–13
Représentation hiérarchique du modèle d'une structure d'assemblage

Ce schéma est conforme à la règle du polymorphisme : des éléments dont les attributs portent la sémantique structurent les niveaux. La DTD de ce schéma est la suivante :

```
<!ELEMENT meta-structure (status, heading+)>
<!ELEMENT status (meta*)>
<!ELEMENT meta (name, value)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT heading (title?, subtitle?, acronym?, revision?,
(heading|dm)*)>
```

```
<!ATTLIST heading type (front|body|rear|other) "other">
<!ELEMENT dm (title?, filename)>
<!ELEMENT title     (#PCDATA)>
<!ELEMENT filename (#PCDATA)>
<!ELEMENT subtitle (#PCDATA)>
<!ELEMENT acronym   (#PCDATA)>
<!ELEMENT revision (#PCDATA)>
```

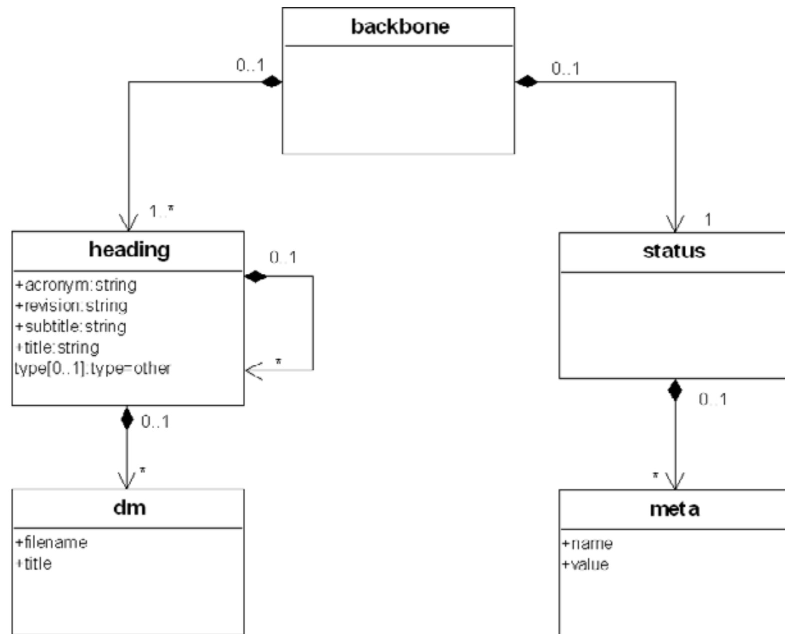


Figure 10–14
Diagramme UML de la structure d'assemblage

Exemple concret

Le modèle que nous allons prendre comme exemple est une référence historique dans le domaine de la modularisation. Il s'agit de la norme S1000D, utilisée pour les données de maintenance. La norme s'applique autant à des documents qu'à des données. À ce titre, elle est naturellement modulaire : les opérations de maintenance telles que le montage/démontage d'un système sont souvent communes à plusieurs appareils.

Nous verrons dans cette section comment les modèles définis par la norme sont eux-mêmes modularisés. En effet, la S1000D ne se contente pas de définir une documentation sous la forme de modules d'information, mais est elle-même constituée de modèles découpés en modules. La modularité est totale.

Un module conforme à la S1000D est un document XML qui comporte deux parties : l'une contient les données de gestion et d'applicabilité du contenu, l'autre le contenu à proprement parler. La spécification sépare nettement le vocabulaire XML destiné à la structuration du fonds documentaire de celui destiné à sa gestion.

Identification des modules

Dans la norme S1000D, un module est défini comme étant « *une unité d'information autosuffisante, contenant un texte et des illustrations relatifs à la mise en œuvre et à la maintenance d'un aéronef, d'un équipement ou d'un matériel de soutien* ». Cette unité d'information est réalisée de telle manière qu'elle puisse être intégrée et retrouvée dans une base de données à partir de son *Module Code* (DMC ou Data Module Code). Cet identifiant est bien plus qu'une simple suite unique de caractères alphabétiques : le DMC est une véritable plaque d'immatriculation du module et traduit, à lui tout seul, son plan de classement par rapport à la logique des opérations de maintenance. Le tableau 10-4 décrit la structure d'un identifiant de module : la lettre a y représente un caractère alphabétique, 1 un caractère numérique et α un caractère alphanumérique. Des exemples réels sont donnés juste après.

Tableau 10–4 Structure lexicale du code d'identification d'un module

Code	Format	Signification du code
MI	αα	Projet
SDC	A	Configuration
SNS	αα -11- αα	Sujet
DC	α1	Numéro de l'opération de maintenance concernée
DCV	a	
IC	111	Type d'information contenue dans le module
ICV	a	
ILC	a	Localisation de l'opération de maintenance

Tableau 10–5 Exemples réels de codes d'identification de modules

Data Module Code	Sujet technique	Information
A1-A-21-23-41-00A-030A-C	Prise de conditionnement	Données techniques
A1-A-21-23-41-00A-040A-C	Prise de conditionnement	Description physique et fonctionnelle
A1-A-21-23-41-00A-800A-C	Prise de conditionnement	Procédures et informations de stockage
A1-A-21-33-22-00A-030A-D	Raccord mobile pressurisation radar	Données techniques
A1-A-21-33-22-00A-040A-D	Raccord mobile pressurisation radar	Description physique et fonctionnelle
A1-A-21-33-22-00A-800A-C	Raccord mobile pressurisation radar	Procédures et informations de stockage

Tableau 10–5 Exemples réels de codes d'identification de modules

Data Module Code	Sujet technique	Information
A1-A-21-51-11-00A-030A-D	Échangeur primaire	Données techniques
A1-A-21-51-11-00A-040A-D	Échangeur primaire	Description physique et fonctionnelle
A1-A-21-51-11-00A-800A-C	Échangeur primaire	Procédures et informations de stockage
A1-A-21-52-11-00A-030A-D	Échangeur principal	Données techniques
A1-A-21-52-11-00A-040A-D	Échangeur principal	Description physique et fonctionnelle
A1-A-21-52-11-00A-520A-C	Échangeur principal	Dépose
A1-A-21-52-11-00A-720A-C	Échangeur principal	Pose

La structure XML correspondante montre que l'identifiant est saisi au moyen de treize balises XML (l'élément `dmc` et les douze balises qu'il contient) :

```
<dmc>
<avee>
<modelic>1B</modelic>
<sdc>A</sdc>
<chapnum>31</chapnum>
<section>1</section>
<subsect>5</subsect>
<subject>01</subject>
<discode>00</discode>
<discodev>A</discodev>
<incode>520</incode>
<incodev>A</incodev>
<itemloc>A</itemloc>
</avee>
</dmc>
```

Classification des modules

Chaque module a un contenu dont la structure dépend du type de l'information contenue. Le tableau 10-6 montre la classification établie dans le cadre de notre exemple.

Tableau 10–6 Classification des modules S1000D

Description	Chapitres/sections/sous-sections, paragraphes/listes/figures…
Procédure	Une série d'étapes (élément `step`) qui correspondent aux différentes tâches à exécuter lors des opérations de maintenance telles que montage/démontage.
Catalogue illustré	Tableaux d'équipements et de composants et figures associées.

Tableau 10–6 Classification des modules S1000D

Manuel de vol	Une partie descriptive associée à des *check-lists*.
Câblage	Définition des câbles et connecteurs électriques.
Planification	Liste des procédures de maintenance avec leur périodicité d'exécution (ex. vidange tous les 5000 km) et les potentiels des équipements (limites de vie, stockage…).
Panne	Succession des actions nécessaires au diagnostic d'une panne (si... alors… sinon…).

On voit ici comment des modules, après avoir été identifiés structurellement par rapport à une arborescence XML, peuvent encore faire l'objet d'une classification en fonction de la nature des informations qu'ils contiendront.

Structure d'assemblage

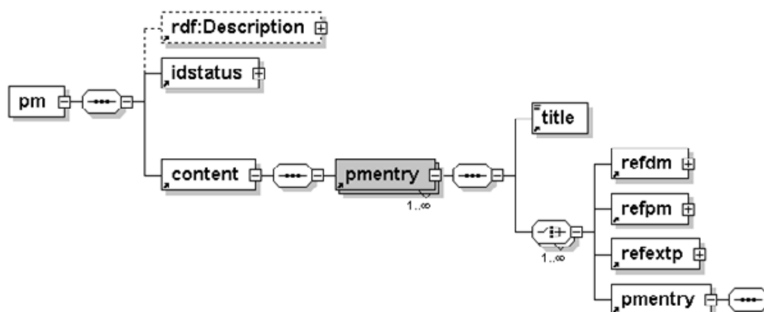
Les structures d'assemblage sont des documents XML qui servent à monter une publication : ils spécifient l'ordre des modules à assembler et donnent les identifiants des modules à utiliser.

La figure 10-15 montre la structure d'assemblage employée par la S1000D.

L'élément racine est `pm`, pour *publication module*. On remarque que les éléments `idstatus` et `content` sous `pm` sont exactement les mêmes que ceux utilisés dans tous les autres cas de modules S1000D : du point de vue de la gestion, une publication est un module comme les autres.

La branche qui nous intéresse particulièrement ici est `content`. Elle joue en effet le rôle d'élément purement structurel ; ce dernier est immédiatement suivi d'un élément répétable et récursif : `pmentry`. Un tel modèle de structure est particulièrement typique des structures d'assemblage.

Figure 10–15
Structure d'assemblage des modules de la S1000D



Chaque niveau de la structure d'assemblage possède, à la manière des tables des matières, son propre titre (l'élément `title`), suivi de quatre possibilités : la référence

à un module (refdm), la référence à une autre structure d'assemblage (refpm), un lien vers une autre documentation technique (refextp), et enfin l'ouverture d'un sous-niveau (pmentry).

Les modules inclus sont référencés au moyen de leur code DMC (Data Module Code).

Nous fournissons enfin un exemple concret de document XML représentant une structure d'assemblage valide par rapport à ce modèle :

```
<?xml version="1.0"?>
<pm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.s1000d.org/S1000D_2-0/
    xml_schema/pm/pmSchema.xsd"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://www.purl.org/dc/elements/1.1/"
    xmlns:xlink="http://www.w3.org/1999/xlink">
<idstatus>
  <pmaddres>
<!-- Code de la structure d'assemblage -->
<!-- Cela afin de pouvoir la référencer d'une autre structure
d'assemblage -->
   <pmc>
    <modelic>1B</modelic>
    <pmissuer>D9460</pmissuer>
    <pmnumber>00001</pmnumber>
    <pmvolume>00</pmvolume>
   </pmc>
<!-- Métadonnées de la structure d'assemblage : elle-même est gérée
comme un module -->
   <pmtitle>List of Applicable Publications - Eurofighter</pmtitle>
   <issno issno="002" type="changed"></issno>
   <issdate year="2003" month="09" day="04"></issdate>
  </pmaddres>
  <pmstatus>
   <security class="02"></security>
   <rpc>C0419</rpc>
   <media type="CD-ROM" code=" DSK: 1B-A/LOAP-00-D"></media>
   <qa>
    <firstver type="tabtop"></firstver>
   </qa>
  </pmstatus>
</idstatus>
<content>
<!-- Ouverture d'un niveau assimilable à un niveau de section -->
<pmentry>
<!-- Titre de ladite section -->
```

```
  <title>Front Matter</title>
  <refdm>
<!-- Code d'un module composant la publication -->
   <dmc>
    <avee>
     <modelic>1B</modelic>
     <sdc>A</sdc>
     <chapnum>00</chapnum>
     <section>4</section>
     <subsect>0</subsect>
     <subject>00</subject>
     <discode>00</discode>
     <discodev>A</discodev>
     <incode>001</incode>
     <incodev>A</incodev>
     <itemloc>A</itemloc>
    </avee>
   </dmc>
   <issno issno="001" type="new"></issno>
  </refdm>
  <refdm>
<!-- Code d'un module composant la publication au même niveau
hiérarchique-->
   <dmc>
   … … … … …
   </dmc>
  </refdm>
</pmentry>
<!-- Ouverture d'un niveau assimilable à un niveau de section -->
<pmentry>
<!-- Titre de ladite section -->
  <title>Introduction</title>
<!-- Code d'un module composant la publication -->
   … … … …
</pmentry>
<!-- Ouverture d'un niveau assimilable à un niveau de section -->
  <pmentry>
   … … … … …
<!-- Ouverture d'un nouveau niveau dans la publication -->
   <pmentry>
    … … … … …
   </pmentry>
  </pmentry>
</content>
</pm>
```

On y remarquera particulièrement les points suivants :

- La structure d'assemblage est gérée comme un module : elle a son propre jeu de métadonnées de gestion.
- Le découpage de type chapitre/section est assuré par une balise récursive (`pmentry`) ; chaque section a son propre titre, autonome et indépendant de celui contenu dans les modules.
- Chaque module composant la publication est référencé non par un lien simple, mais par une structure XML de type complexe.

En résumé

Ce chapitre a permis d'aborder de nombreux points relatifs à la mise en œuvre du concept de publication modulaire. Nous avons montré en particulier :

- la manière de définir la taille d'un module d'information,
- l'importance de la présence de clés de voûte dans les schémas,
- la manière de décrire dans les schémas XML des éléments jouant le rôle d'articulations,
- les règles de conception de documents modulaires.

Les règles que nous venons de passer en revue tiennent à la fois du bon sens et des contraintes intrinsèques de XML. Nous avons notamment insisté sur les principes et règles suivants :

- Le seul découpage d'un modèle XML au niveau des éléments purement structurels ne suffit pas à déterminer la bonne taille des modules.
- Le découpage en modules n'est pas un débitage du schéma XML.
- Une publication est le résultat de deux mondes : celui des données et celui des traitements.
- Les modules ne doivent pas être emboîtés les uns dans les autres.
- Les critères d'applicabilité doivent s'exprimer simplement et s'appliquer par famille à des modules entiers.
- Le balisage doit être polymorphe tant à l'intérieur des modules que dans les structures d'assemblage.
- Des règles d'indépendance et de neutralité s'appliquent aux identifiants qui servent aux liens.
- Les métadonnées sont une interface entre le contenu du module et le système qui le gère.

En passant de la donnée au module, puis à la publication, et en étant capable, grâce aux règles présentées dans ce chapitre, de hiérarchiser ces trois niveaux d'information, on répond en grande partie à l'une des exigences de base de la gestion des connaissances, exprimée ainsi par le professeur Shigehisa Tsuchiya :

« Bien que les termes données, information et connaissance soient souvent pris les uns pour les autres, il existe une distinction claire entre eux. Quand on donne du sens à la donnée via un cadre interprétatif, elle devient information, et quand on lit une information en lui donnant du sens via un cadre interprétatif, elle devient de la connaissance. »

11

Modèles pour la gestion des métadonnées

Abordons maintenant le sujet des métadonnées, lesquelles fournissent des informations sur la nature des autres données. Sur cette question, le cas de XML est particulièrement remarquable. Selon ce que l'on met derrière le vocable de métadonnées, on peut repérer au moins trois niveaux concernés dans XML : les niveaux du modèle, du document, et enfin des données contenues à l'intérieur du document.

En XML, les métadonnées sont des informations qui se situent à la fois dans le domaine de la gestion et dans celui du contenu. En effet, les métadonnées peuvent être aussi bien utilisées pour typer les documents eux-mêmes que les données qu'ils contiennent.

Il est possible de définir son propre jeu de métadonnées, à la manière de la norme S1000D, comme d'utiliser un modèle tout fait, tel Dublin Core. Enfin, il existe un modèle standard générique pour définir des modèles de métadonnées : c'est la recommandation RDF du W3C.

Nous verrons, dans les prochaines sections :

- les métadonnées définies par le schéma XML ;
- les métadonnées relatives à l'entité document ;
- les métadonnées contenues dans le corps du document.

Métadonnées définies par le schéma XML

Au niveau du modèle, et particulièrement avec XML Schema, les métadonnées sont explicites puisqu'il s'agit des noms des types utilisés. On connaît ainsi la nature exacte des données contenues dans les éléments et les attributs. Par exemple, le type `xs:date`, ou l'un quelconque de ses dérivés, indique que la donnée correspondante est une date et non une quelconque chaîne de caractères.

En XML toutefois, la notion de métadonnées ne se limite pas au seul type. Il existe d'autres informations telles que l'espace de noms auquel appartient la donnée, les éventuelles valeurs par défaut, les formes lexicales de base, canonique, logique, etc.

Voilà pourquoi une forme de document XML a été inventée : le PSVI, pour Post Schema Validation Infoset. Elle est constituée du document XML de base, augmenté de toutes les métadonnées issues du schéma XML qui lui sert de modèle. Il ne s'agit pas d'une juxtaposition du document XML et de son modèle, mais véritablement d'une symbiose, comme nous allons le voir dans l'exemple suivant.

Voici un document XML volontairement succinct :

```
<?xml version="1.0" encoding="UTF-8"?>
<maison xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="chapitre11.xsd">
 <murs>4</murs>
 <toit>un toit</toit>
</maison>
```

et le schéma XML qui le valide :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
 <xs:simpleType name="nonNegativeDecimal">
  <xs:restriction base="xs:decimal">
   <xs:minExclusive value="0"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:element name="maison">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="murs" type="nonNegativeDecimal"/>
    <xs:element name="toit" type="xs:string"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Voici maintenant le PSVI de ce document XML, dans lequel vous remarquerez que :

- Le vocabulaire des éléments et attributs n'est ni celui du document XML ni celui de son schéma. C'est un vocabulaire propre à l'expression des métadonnées.
- Les structures définies par le schéma XML ne sont pas explicites. Un PSVI n'est pas destiné à exprimer tous les modèles de contenu d'un vocabulaire XML ; cela reste le rôle des schémas.
- Les données initiales sont enrichies de celles produites par le parseur, lesquelles indiquent les conditions de validité des éléments, attributs et données : le contexte et le type de validation.

Une lecture même rapide d'un PSVI aide à comprendre qu'il est possible d'écrire un programme universel de lecture d'un document XML et de toutes les données qui l'accompagnent. Cependant, un tel programme serait incapable de récrire le schéma d'origine, ce n'est pas dans ses attributions.

Dans l'extrait suivant, nous avons encadré et commenté les parties qui nous semblent significatives.

```
<document xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
          xmlns:psv='http://apache.org/xml/2001/PSVInfosetExtension'
          xmlns='http://www.w3.org/2001/05/XMLInfoset'>
 <characterEncodingScheme>UTF-8</characterEncodingScheme>
 <standalone xsi:nil='true'/>
 <version>1.0</version>
 <children>
  <element>
   <namespaceName xsi:nil='true'/>
   <!-- maison est l'élément racine du document XML originel -->
   <localName>maison</localName>
   <prefix xsi:nil='true'/>
<attributes>
   <!-- métadonnées de tous les attributs de l'élément maison -->
 <attribute>
  <namespaceName>http://www.w3.org/2001/XMLSchema-instance
  </namespaceName>
  <localName>noNamespaceSchemaLocation</localName>
  <prefix>xsi</prefix>
  <normalizedValue>chapitre11.xsd</normalizedValue>
  <attributeType xsi:nil='true'/>
  <references xsi:nil='true'/>
  <psv:validationAttempted>full</psv:validationAttempted>
  <psv:validationContext>maison</psv:validationContext>
  <psv:validity>valid</psv:validity>
  <psv:schemaErrorCode></psv:schemaErrorCode>
```

```
   <psv:schemaNormalizedValue>chapitre11.xsd</psv:schemaNormalizedValue>
   <psv:schemaSpecified>schema</psv:schemaSpecified>
   <psv:typeDefinitionType>simple</psv:typeDefinitionType>
   <psv:typeDefinitionNamespace>http://www.w3.org/2001/XMLSchema
   </psv:typeDefinitionNamespace>
   <psv:typeDefinitionAnonymous>false</psv:typeDefinitionAnonymous>
   <psv:typeDefinitionName>anyURI</psv:typeDefinitionName>
  </attribute>
 </attributes>
    <namespaceAttributes>
  <attribute>
   <namespaceName>http://www.w3.org/2000/xmlns/</namespaceName>
   <localName>xsi</localName>
   <prefix>xmlns</prefix>
   <normalizedValue>http://www.w3.org/2001/XMLSchema-instance
   </normalizedValue>
   <specified>true</specified>
   <attributeType>CDATA</attributeType>
   <references xsi:nil='true'/>
   <psv:validationAttempted>none</psv:validationAttempted>
   <psv:validationContext>maison</psv:validationContext>
   <psv:validity>unknown</psv:validity>
   <psv:schemaErrorCode></psv:schemaErrorCode>
   <psv:schemaNormalizedValue xsi:nil='true'/>
   <psv:schemaSpecified>schema</psv:schemaSpecified>
  </attribute>
    </namespaceAttributes>
    <psv:validationContext>maison</psv:validationContext>
    <psv:typeDefinitionType>complex</psv:typeDefinitionType>
    <psv:typeDefinitionNamespace xsi:nil='true'/>
    <psv:typeDefinitionAnonymous>true</psv:typeDefinitionAnonymous>
    <psv:typeDefinitionName xsi:nil='true'/>
    <children>
  <element>
   <namespaceName xsi:nil='true'/>
    <!-- murs est l'un des deux sous-éléments de l'élément maison -->
   <localName>murs</localName>
   <prefix xsi:nil='true'/>
   <attributes/>
   <namespaceAttributes/>
   <psv:validationContext>maison</psv:validationContext>
   <psv:typeDefinitionType>simple</psv:typeDefinitionType>
   <psv:typeDefinitionNamespace xsi:nil='true'/>
   <psv:typeDefinitionAnonymous>false</psv:typeDefinitionAnonymous>
   <psv:typeDefinitionName>nonNegativeDecimal</psv:typeDefinitionName>
```

```
 <children>
 </children>
 <inScopeNamespaces>
  <namespace>
   <prefix>xml</prefix>
   <namespaceName>http://www.w3.org/XML/1998/namespace</namespaceName>
  </namespace>
  <namespace>
   <prefix>xmlns</prefix>
   <namespaceName>http://www.w3.org/2000/xmlns/</namespaceName>
  </namespace>
  <namespace>
   <prefix>xsi</prefix>
   <namespaceName>http://www.w3.org/2001/XMLSchema-instance
   </namespaceName>
  </namespace>
 </inScopeNamespaces>
 <psv:validationAttempted>full</psv:validationAttempted>
 <psv:validity>valid</psv:validity>
 <psv:schemaErrorCode xsi:nil='true'/>
  <!–Ici on retrouve le contenu de l'élément murs -->
 <psv:schemaNormalizedValue>4</psv:schemaNormalizedValue>
 <psv:schemaSpecified>schema</psv:schemaSpecified>
</element>
<element>
 <namespaceName xsi:nil='true'/>
  <!–Ici on retrouve l'élément toit -->
 <localName>toit</localName>
 <prefix xsi:nil='true'/>
 <attributes/>
 <namespaceAttributes/>
 <psv:validationContext>maison</psv:validationContext>
 <psv:typeDefinitionType>simple</psv:typeDefinitionType>
 <psv:typeDefinitionNamespace>http://www.w3.org/2001/XMLSchema
 </psv:typeDefinitionNamespace >
 <psv:typeDefinitionAnonymous>false</psv:typeDefinitionAnonymous>
 <psv:typeDefinitionName>string</psv:typeDefinitionName>
 <children>
 </children>
 <inScopeNamespaces>
  <namespace>
   <prefix>xml</prefix>
   <namespaceName>http://www.w3.org/XML/1998/namespace</namespaceName>
  </namespace>
  <namespace>
```

```
    <prefix>xmlns</prefix>
    <namespaceName>http://www.w3.org/2000/xmlns/</namespaceName>
   </namespace>
   <namespace>
    <prefix>xsi</prefix>
    <namespaceName>http://www.w3.org/2001/XMLSchema-instance
    </namespaceName>
   </namespace>
  </inScopeNamespaces>
  <psv:validationAttempted>full</psv:validationAttempted>
  <psv:validity>valid</psv:validity>
  <psv:schemaErrorCode xsi:nil='true'/>
   <!—Ici on retrouve le contenu de l'élément toit -->
  <psv:schemaNormalizedValue>un toit</psv:schemaNormalizedValue>
  <psv:schemaSpecified>schema</psv:schemaSpecified>
 </element>
   </children>
   <inScopeNamespaces>
 <namespace>
  <prefix>xml</prefix>
  <namespaceName>http://www.w3.org/XML/1998/namespace</namespaceName>
 </namespace>
 <namespace>
  <prefix>xmlns</prefix>
  <namespaceName>http://www.w3.org/2000/xmlns/</namespaceName>
 </namespace>
 <namespace>
  <prefix>xsi</prefix>
  <namespaceName>http://www.w3.org/2001/XMLSchema-instance
  </namespaceName>
 </namespace>
   </inScopeNamespaces>
   <psv:validationAttempted>full</psv:validationAttempted>
   <psv:validity>valid</psv:validity>
   <psv:schemaErrorCode xsi:nil='true'/>
   <psv:schemaNormalizedValue xsi:nil='true'/>
   <psv:schemaSpecified>schema</psv:schemaSpecified>
  </element>
 </children>
 <documentElement xsi:nil='true'/>
 <notations/>
 <unparsedEntities/>
 <baseURI xsi:nil='true'/>
 <allDeclarationsProcessed>true</allDeclarationsProcessed>
</document>
```

L'expression des métadonnées *via* le PSVI exige un vocabulaire précis, reconnu de toutes les applications. Le vocabulaire du PSVI est en effet indépendant de toute spécification métier : il ne fait que fournir des informations par rapport au seul monde XML. Dans les sections suivantes, nous verrons comment en affiner les possibilités afin de transmettre des métadonnées de manière concise.

Métadonnées relatives à l'entité document

Certaines métadonnées apportent des informations sur le document en tant qu'entité. Celles-là ont donc pour vocation essentielle de servir à gérer le document XML en tant qu'enveloppe.

Le vocabulaire utilisé pour transmettre ces métadonnées peut être :

- spécifique (nous verrons le cas de la norme métier S1000D),
- générique (un exemple typique est celui de HTML),
- commun (nous verrons le cas du Dublin Core),
- sophistiqué (nous présenterons RDF).

Au travers d'exemples pratiques, nous allons étudier ces différents cas de figure.

Métadonnées spécifiques d'une application métier : cas de la S1000D

Le cas de la norme S1000D a été introduit au chapitre 8. Nous avons en particulier expliqué comment cette norme définissait plus d'une quinzaine de modèles à partir d'un ensemble de schémas génériques combinables. Nous avons notamment montré que le méta-modèle logique de la S1000D était organisé autour de trois couches de schémas. Cette organisation était représentée schématiquement à la figure 8-1.

Les schémas du niveau de la couche 0 sont communs à tous les modèles. C'est dans cette couche que se trouve le modèle définissant les métadonnées des modules de la S1000D. Remarquant cela, on en déduit que les concepteurs de la S1000D ont souhaité que tous les modules puissent être gérés dans un seul et même système de gestion : ce n'est pas un mal !

La structure définie pour les métadonnées est composée uniquement d'éléments et de quelques attributs XML spécifiques à l'application S1000D. Elle est représentée à la figure 11-1.

Le modèle montre que ses métadonnées (`idstatus`) sont nettement séparées du contenu même du module (qui est entièrement compris dans l'élément `content`) auquel elles servent, en quelque sorte, de prologue.

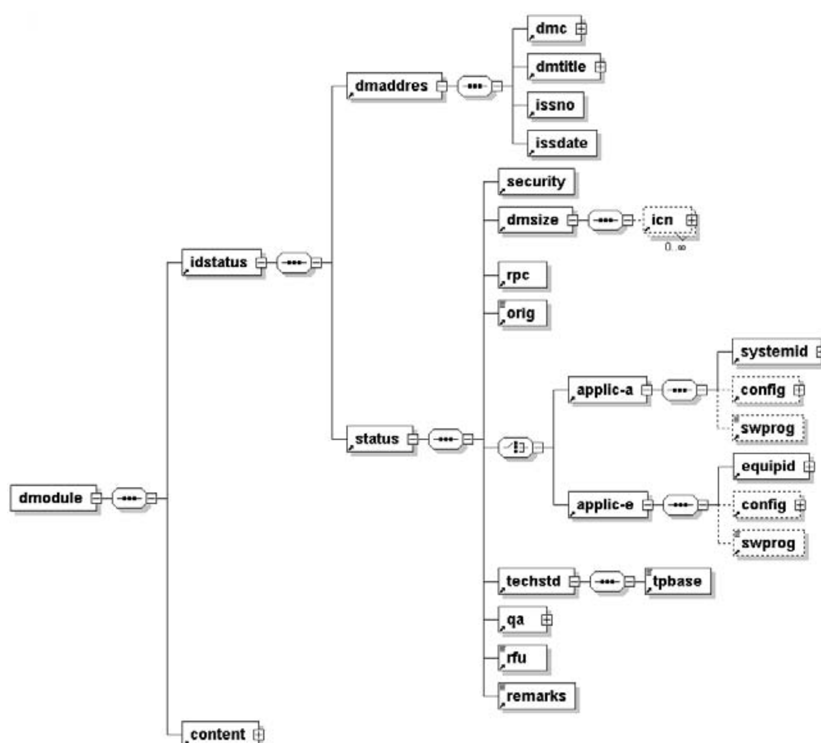


Figure 11–1
Données de gestion d'un module S1000D

Les métadonnées elles-mêmes sont constituées de deux parties distinctes. L'une, `dmaddres`, contient le code d'identification du module (élément `dmc` comme *data module code*), son titre externe (élément `dmtitle`), ses numéro et date d'édition (`issno` et `issdate`). L'autre, `status`, renferme des informations générales de gestion et de contrôle (dont celles concernant la taille du module, son statut éditorial et son émetteur), ainsi que les critères de son applicabilité qui, dans notre cas présent, peut être un modèle d'avion (`applic-a`) ou d'équipement (`applic-e`).

Bien qu'initialement prévu pour les seuls avions militaires, ce modèle a inspiré de nombreux concepteurs de schémas XML.

Voici par exemple une instance réelle de ce modèle :

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE dmodule PUBLIC "-//Aecma//DTD Aecma 1000D Procedural
20010401//EN" "procedx.dtd" []>
<dmodule>
<idstatus>
<!-- Partie identification -->
```

```
<dmaddres>
<dmc>
<avee>
<!-- Identification du module -->
<modelic>1B</modelic><sdc>A</sdc><chapnum>31</chapnum><section>1
</section>
<subsect>6</subsect><subject>00</subject><discode>00</discode>
<discodev>A</discodev><incode>040</incode><incodev>A</incodev>
<itemloc>A</itemloc>
</avee>
</dmc>
<!-- Titres du module -->
<dmtitle>
<techname>Multifunction Head Down Display, LH</techname>
<infoname>Remove Procedures</infoname>
</dmtitle>
<!—Numéro et date d'émission -->
<issno issno="004" type="changed"/>
<issdate year="2001" month="05" day="31"/>
</dmaddres>
<!-- Partie status -->
<status>
<security class="1"/>
<dmsize>4 Pages</dmsize>
<rpc>I9005</rpc>
<orig>Mikel.Kel</orig>
<applic-a>
 <systemid>
  <version versid=" VER " from="1" to="2"/>
 </systemid>
 <config>
  <mod modtyp="prandpo" modnb="module-number" modcond="engmod"/>
 </config>
</applic-a>
<techstd>
<authblk></authblk>
<authex></authex>
<notes></notes>
</techstd>
<qa><unverif/></qa>
<sbc>X1BA311501000</sbc>
<skill skill="b"/>
<remarks>None</remarks>
</status>
</idstatus>
```

L'identification du module repose sur un élément de type complexe, l'élément `dmc`, un titre `dmtitle` constitué d'un sujet (`techname`) et d'une description en clair du type de l'information contenue dans le module (`infoname`). Dans notre exemple, ce type est une procédure de démontage.

Dans les informations générales de gestion, on relève la présence des éléments `security`, `orig` et `rpc` qui contiennent respectivement le niveau de confidentialité du module et les identifiants des personnes morales responsables de la rédaction du module et de sa fourniture.

L'applicabilité est gérée par l'élément `applic`. Nous avons consacré une section du chapitre 10 à ce sujet.

On remarquera enfin les éléments relatifs à la qualité `qa`, aux liens qui sont une particularité de l'arborescence physique ou fonctionnelle du système documenté (les éléments `sbc` et `fic` représentent respectivement le code de décomposition système, ou *system breakdown code*, et le code d'item fonctionnel ou *Functional Item Code*). On y trouve également les niveaux de qualification requis pour exécuter la procédure (élément `skill`) ou encore relatifs aux raisons qui ont provoqué la révision du module (élément `rfu` qui signifie *reason for update*). On notera qu'en ce qui concerne la qualité, le contenu de l'élément `qa` est imposé par un mécanisme de choix entre deux éléments vides : `verif` (« le module a été vérifié ») et `unverif` (« le module n'a pas été vérifié »). Point de liberté d'écriture à ce niveau : l'auteur du module doit choisir l'un ou l'autre de ces deux éléments.

Que faut-il en conclure ?

Ces métadonnées appartiennent à quatre catégories différentes :

- La catégorie des métadonnées de *rangement* du module : son *data module code* qui, remarquons-le, est une structure complexe calquée sur une décomposition plus générale (métier) du système documenté.
- La catégorie des métadonnées de *description* générale du contenu du module : son titre externe (pour le retrouver par son titre) et la nature de l'information qu'il contient.
- La catégorie des informations de gestion, ou métadonnées de *gestion* : numéro de révision, date d'édition, émetteur, statut éditorial.
- Enfin la catégorie des informations destinées aux utilisateurs, ou métadonnées *applicatives* : applicabilité, niveau de confidentialité, taille du module en nombre de pages.

On peut dire que ces catégories sont bien représentatives des métadonnées qui devront le plus souvent être prises en compte.

Métadonnées spécifiques d’un média particulier : cas de xHTML

Le cas de xHTML est intéressant à double titre : d’une part, la structure dédiée aux métadonnées est des plus simples et d’autre part, le problème de la recherche de pages Web est à l’origine de plusieurs travaux sur les métadonnées. C’est en effet en vue de rendre la recherche de page Web plus efficace que les Dublin Core et autres normes RDF sont étudiées.

Dans cette section, nous allons étudier le mécanisme de base de xHTML concernant les métadonnées.

Dans les pages xHTML, c’est simple, il n’existe qu’un seul élément pour déclarer des métadonnées : c’est l’élément `meta`. Il est certes normalisé, mais de telle manière que l’utilisateur est totalement libre de faire passer la sémantique qu’il veut. Cet élément est équipé de deux attributs : l’un de nom `name` et l’autre de nom `content`. Le premier sert à transmettre à l’application un nom de métadonnée et l’autre sa valeur. On peut ainsi créer autant de couples (`nom`,`valeur`) que nécessaire à partir du seul élément `meta`.

Dans l’exemple de la figure 11-2, nous donnons une représentation du début d’un document xHTML dans lequel :

- On repère facilement l’élément `meta`.
- La feuille de styles utilisée présente le contenu des attributs `name` et `content` comme s’il s’agissait de texte normal.

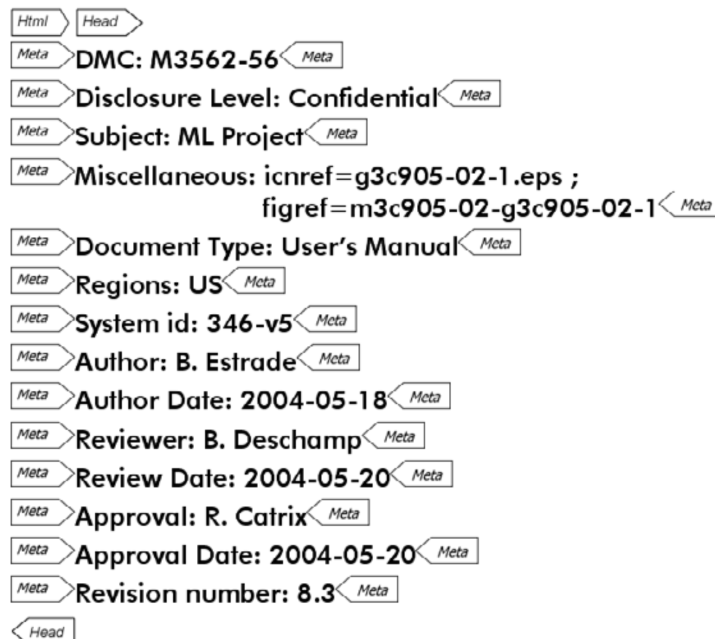


Figure 11–2
Exemples de métadonnées dans l’en-tête d’un document xHTML

Voici la forme XML de cet exemple :

```
<html>
<head>
<title> ML project</title>
<meta name="DMC" content="M3562-56"/>
<meta name="Disclosure Level" content="Confidential"/>
<meta name="Subject" content="ML project"/>
<meta name="Miscellaneous"
      content="icnref=g3c905-02-1.eps;figref=m3c905-02-g3c905-02-1"/>
<meta name="Document Type" content="User's Manual"/>
<meta name="Region" content="US"/>
<meta name="System id" content="346-v5"/>
<meta name="Author" content="B. Estrade"/>
<meta name="reviewer" content="B. Deschamps"/>
<meta name="review Date" content="2004-05-20"/>
<meta name="Approval" content="R. Catrix"/>
<meta name="Approval Date" content="2004-05-20"/>
<meta name="Revision number" content="8.3"/>
</head>
```

Cette approche présente les avantages suivants :

- Simplicité de mise en œuvre.
- Ouverture : il n'y a aucune limite et aucune contrainte dans le choix des noms de métadonnées.
- Compatibilité HTML : quand un tel modèle est utilisé dans un document XML, sa transposition en HTML est des plus simples.

Inconvénients :

- La structure de l'élément `meta` est plate, ce qui ne permet pas d'avoir une grande précision : il n'est pas possible de grouper les métadonnées en paquets logiques.
- Il n'est pas possible d'établir des liens entre les métadonnées.
- Il n'est pas (ou difficilement) possible de contrôler les mots utilisés comme valeur des métadonnées : des erreurs peuvent se glisser lors de leur saisie.
- Il n'est pas possible de spécifier le type de l'attribut `content` : ce ne peut être qu'une chaîne de caractères quelconque.

Que faut-il en conclure ?

Ce modèle est souple, flexible, immédiatement opérationnel, et pourra à ce titre satisfaire de nombreux besoins. Le manque de contrôle est son principal problème. Ainsi, ce modèle est insuffisant pour ceux qui doivent manipuler des informations sensibles comme c'est le cas avec la S1000D.

Nous verrons dans la section suivante un modèle intermédiaire qui a l'avantage de définir un vocabulaire de base : le Dublin Core. Plus avant dans ce chapitre, nous traiterons d'un modèle très sophistiqué avec le modèle RDF : ce dernier ne propose ni vocabulaire ni structure prédéfinie, mais offre des mécanismes pour définir une structure et un vocabulaire.

Métadonnées définies par le Dublin Core

Le Dublin Core est composé d'un ensemble fini d'éléments dont la sémantique a été établie une fois pour toutes.

VOCABULAIRE Dublin Core

Le Dublin Core se dénomme ainsi parce que la première réunion du groupe de travail sur le sujet se tint à Dublin. C'était en 1995. Le groupe de travail, quant à lui, fut constitué en octobre 1994.

Dès la création du groupe de travail sur Dublin Core, les objectifs étaient de mettre en place un modèle améliorant les possibilités de recherche d'information sur le Web. C'était en 1994 et le Web naissait à peine !

Les éléments du Dublin Core expriment directement la sémantique des métadonnées alors que ceux de modèles plus sophistiqués, tels que RDF qui sera vu dans une prochaine section, traitent des aspects structuraux des descripteurs (ce qui revient à codifier la sémantique des ressources identifiées par les descripteurs). En d'autres termes, Dublin Core est un vocabulaire figé et fini de noms d'éléments, tandis que RDF permet de créer son propre vocabulaire d'éléments et d'attributs.

Le Dublin Core définit un jeu de 15 éléments pour identifier une information. Il est largement utilisé dans le monde éditorial et a été retenu pour une partie des métadonnées de la S1000D (pour celle qui est compatible avec lui bien évidemment). Les métadonnées de la S1000D concernées par DC (acronyme usuel de Dublin Core) sont celles contenues dans les éléments `identification` et `status`.

Une conséquence immédiate en est que toute application mettant en œuvre Dublin Core pourra classer et gérer *a minima* les modules ainsi caractérisés.

Nous présentons au tableau 11-1 les 15 éléments du Dublin Core et faisons suivre ce tableau d'un exemple concret de mise en œuvre.

Tableau 11–1 Éléments de base du Dublin Core et leur signification

Éléments	Signification
<dc:title>	Titre donné à la ressource pour la reconnaître.
<dc:creator>	Personne physique ou morale responsable de la fabrication du contenu de la ressource.

Tableau 11–1 Éléments de base du Dublin Core et leur signification

Éléments	Signification
<dc:subject>	Une phrase, des mots-clés, un code de classification ou toute autre donnée textuelle aidant à décrire le sujet du contenu de la ressource.
<dc:description>	Un résumé, une table des matières, une référence à une représentation graphique ou toute autre forme de description textuelle du contenu de la ressource.
<dc:publisher>	Personne physique ou morale en charge de l'édition (mise en forme et diffusion) de la ressource.
<dc:contributor>	Personne physique ou morale qui contribue à la fabrication du contenu de la ressource.
<dc:date>	Une date qui est généralement la date de création ou de publication de la ressource. Il est recommandé que cette date suive le format YYYY-MM-DD défini dans le profil ISO 8601 [W3CDTF].
<dc:type>	La nature ou le genre du contenu de la ressource. En général, il est recommandé ici d'utiliser les mots d'un vocabulaire de classification reconnu.
<dc:format>	Le format du support physique associé à la ressource. Il est recommandé d'utiliser des termes reconnus tels ceux de la liste des types de médias Internet ou MIME.
<dc:identifier>	Identifiant unique (dans un certain contexte) de la ressource. Par exemple, le numéro ISBN pour un livre, les URI et URL pour les ressources électroniques sont de bons candidats. Vous pouvez également utiliser ici vos propres identifiants uniques, tels les codes d'identification des modules (DMC ou Data Module Code) ou d'illustrations (ICN comme Illustration Code Number).
<dc:source>	Référence d'une ressource parente de la ressource présente. Cette référence peut être, par exemple, l'identifiant unique de la ressource parente indiquée par son <dc:identifier>.
<dc:language>	Langue du contenu intellectuel de la ressource. Il est recommandé d'utiliser les codes de langue définis par la norme RFC 3066 combinée avec l'ISO 639. Cela permet de faire des combinaisons de type « fr-FR » pour le français de France et « fr-AKK » pour le français parlé en Acadie.
<dc:relation>	Référence d'une ressource en rapport avec la présente ressource. Il est recommandé d'utiliser ici l'identifiant unique de la ressource en question.
<dc:coverage>	Forme d'applicabilité de la ressource. Cette applicabilité peut être géographique, temporelle ou juridique. Il est recommandé d'utiliser ici des noms, valeurs ou identifiants officiels.
<dc:rights>	Détenteurs des droits de propriété sur le contenu de la ressource. L'information peut être un texte explicite ou une référence à une ressource décrivant les droits de propriété. Ces droits concernent la propriété intellectuelle, le copyright, les conditions d'exploitation, de réutilisation, etc.

En complément à ce jeu de base, des éléments substituables aux premiers permettent de préciser les informations spécifiées. Le tableau 11-2 les présente. La substitution des éléments les uns par les autres est formellement définie dans les schémas XML correspondants (XML Schema dispose d'un mécanisme permettant de définir formellement les éléments autorisés en lieu et place d'autres).

Tableau 11–2 Éléments substituables aux éléments de base du Dublin Core

Elément substituable	Par	Description
<dc:audience>	Nouvel élément	L'audience type de la ressource.
	<dct:mediator>	L'entité type qui gère l'accès à la ressource et pour qui elle est importante (typiquement une bibliothèque).
	<dct:educationLevel>	Niveau de connaissances requis pour les utilisateurs de la ressource.
<dc:coverage>	<dct:spatial>	Caractéristiques spatiales du contenu intellectuel de la ressource.
	<dct:temporal>	Caractéristiques temporelles du contenu intellectuel de la ressource.
<dc:date>	<dct:available>	Date de publication de la ressource.
	<dct:created>	Date de création de la ressource.
	<dct:dateAccepted>	Date de validation ou d'acceptation de la ressource.
	<dct:dateCopyrighted>	Date d'établissement du copyright.
	<dct:dateSubmitted>	Date de soumission de la ressource.
	<dct:medium>	Support physique de la ressource.
	<dct:modified>	Date de modification de la ressource.
	<dct:issued>	Date de sortie formelle de la ressource.
	<dct:valid>	Date de validité de la ressource.
<dc:description>	<dct:abstract>	Résumé du contenu de la ressource.
	<dct:tableOfContents>	Table des matières de la ressource.
<dc:format>	<dct:extent>	Taille ou durée de la ressource.
<dc:identifier>	<dct:bibliographicCitation>	Référence bibliographique.
<dc:relation>	<dct:conformsTo>	Référence d'un standard auquel se conforme la ressource.
	<dct:hasFormat> <dct:isFormatOf>	Relations « a le même format que » et « est du même format que ».
	<dct:hasPart> <dct:isPartOf>	Relations « inclut » et « fait partie de ».
	<dct:hasVersion> <dct:isVersionOf>	Relations « a comme version » et « est une version de ». La notion de version porte ici sur le contenu et non sur le format.
	<dct:references> <dct:isReferencedBy>	Relations « référence » et « est référencé par ».
	<dct:replaces> <dct:isReplacedBy>	Relations « remplace » et « est remplacé par ».
	<dct:requires> <dct:isRequiredBy>	Relations « requiert » et « est requis par ».

Tableau 11–2 Éléments substituables aux éléments de base du Dublin Core

Elément substituable	Par	Description
<dc:rights>	<dct:license>	Document légal spécifiant les droits d'utilisation légaux de la ressource.
	<dct:rightsHolder>	Détenteur des droits.
	<dct:accessRights>	Droits d'accès à la ressource ou indication de son niveau de confidentialité. Par exemple, en cas de restrictions des droits d'accès pour des raisons de liberté individuelle, sécurité ou toute autre loi applicable.
<dc:title>	<dct:alternative>	Titre secondaire de la ressource qui peut être une version traduite du titre ou un titre abrégé.

PRÉCISION Espaces de noms dc et dct

Les éléments de base du Dublin Core et ceux qui leur sont substituables font partie de deux espaces de noms différents. Repérés dans le tableau 11-2 par les préfixes `dc` et `dct`, leurs noms complets sont respectivement `http://purl.org/dc/elements/1.1/` et `http://purl.org/dc/terms/`.

La spécification Dublin Core précise également les normes de codification applicables (telles les normes ISO 3166, 639-1, 639-2, RFC 1766 et 3066, qui spécifient les codes de langues et pays, ainsi que la manière de les combiner pour former des couples langue/pays).

L'exemple suivant montre un cas concret d'utilisation des éléments du Dublin Core, utilisant des éléments des tableaux 11-1 et 11-2.

```
<dc:title>Solenoid Valve, Pump No. 1 (No. 2) - Remove Procedures
</dc:title>
<dc:identifier>1B-B-29-11-01-06A-520A-A_003</dc:identifier>
<dct:isVersionOf>1B-B-29-11-01-06A-520A-A_002</dct:isVersionOf>
<dct:issued>2001-04-01</dct:issued>
<dc:creator>C0419</dc:creator>
<dc:subject>Solenoid Valve, Pump No. 1 (No. 2)</dc:subject>
<dc:type>Remove Procedures</dc:type>
<dc:publisher>C0419</dc:publisher>
<dc:contributor>C0419</dc:contributor>
<dc:date>2001-04-01</dc:date>
<dc:format>text/xml</dc:format>
<dc:language>sx-GB</dc:language>
<dc:rights>1</dc:rights>
<dct:conformsTo>-//Aecma//DTD Aecma 1000D Procedural 20010401//EN
</dct:conformsTo>
```

Enfin, nous donnons ci-après un exemple concret d'utilisation de Dublin Core dans un module S1000D :

```
<?xml version="1.0"?>
<!--PUBLIC "-//S1000D//DTD S1000D Procedural 20030531//EN"-->
<!DOCTYPE dmodule [ … … … …]>
<dmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="procedSchema.xsd"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://www.purl.org/dc/elements/1.1/"
         xmlns:xlink="http://www.w3.org/1999/xlink">
<rdf:Description>
 <dc:Title>Multifunction Head Down Display, LH - Remove Procedures
 </dc:Title>
 <dc:Creator></dc:Creator>
 <dc:Subject>Multifunction Head Down Display, LH - Remove Procedures
 </dc:Subject>
 <dc:Publisher>I9005</dc:Publisher>
 <dc:Contributor></dc:Contributor>
 <dc:Date>2003-05-31</dc:Date>
 <dc:Identifier>1B-A-31-15-01-00A-520A-A_004</dc:Identifier>
 <dc:Rights>01</dc:Rights>
</rdf:Description>
<idstatus>
<dmaddres>
<dmc>
<avee>
<modelic>1B</modelic>
<sdc>A</sdc>
<chapnum>31</chapnum>
<section>1</section>
<subsect>5</subsect>
<subject>01</subject>
<discode>00</discode>
<discodev>A</discodev>
<incode>520</incode>
<incodev>A</incodev>
<itemloc>A</itemloc>
</avee>
</dmc>
<dmtitle>
<techname>Multifunction Head Down Display, LH</techname>
<infoname>Remove Procedures</infoname>
</dmtitle>
```

```
<issno issno="004" inwork="00" type="changed">
</issno>
<issdate year="2003" month="05" day="31">
</issdate>
</dmaddres>
<status>
<security class="01">
</security>
<dmsize>4 Pages</dmsize>
<rpc>I9005</rpc>
<orig></orig>
<applic></applic>
<techstd>
<authblk></authblk>
<authex></authex>
<notes></notes>
</techstd>
<qa>
<unverif></unverif>
</qa>
<sbc>X1BA311501000</sbc>
<skill skill="sk01">
</skill>
<remarks>None</remarks>
</status>
</idstatus>
```

Que remarque-t-on ?

Pour parer aux insuffisances du Dublin Core, certaines données qui faisaient l'objet de plusieurs balises XML dans le modèle originel sont concaténées quand le Dublin Core est utilisé. C'est le cas du code du module, écrit sous la forme `<dc:Identifier>1B-A-31-15-01-00A-520A-A_004</dc:Identifier>`, alors que cela fait l'objet à l'origine des 12 balises contenues dans l'élément `dmc` et de l'attribut `issno` de l'élément `issno`. De même, le titre et la nature de l'information contenue dans le module, les éléments `dmtitle` et `infoname`, sont concaténées et deviennent `<dc:Title>Multifunction Head Down Display, LH - Remove Procedures</dc:Title>`. Enfin, la date, écrite à l'origine sous la forme de trois attributs, devient une chaîne de caractères : `<dc:Date>2003-05-31</dc:Date>`. On comprend dès lors mieux les vraies différences qu'entraîne le choix de l'une ou l'autre approche : les deux sont porteuses d'une même information, mais l'une permet indéniablement de mieux contrôler la qualité de l'information transmise.

Métadonnées définies par RDF

RDF, ou Resource Description Framework, est un modèle de données conçu pour définir des descripteurs de ressources web. RDF est un méta-modèle puisque qu'il sert à définir des modèles de descripteurs. Autant le Dublin Core définit un vocabulaire concret et précis pour spécifier des métadonnées, autant RDF nécessite de définir un vocabulaire et la structure qui l'accompagne. Aussi, le Dublin Core est-il considéré comme une application de RDF.

RDF est le résultat d'un travail réalisé par le W3C pour inventer un langage afin d'apporter de la sémantique aux pages web (plus connu sous le nom de Web sémantique). Son but est d'aboutir à un langage compréhensible par les ordinateurs et les humains. RDF est un « modèle pour modéliser » les métadonnées. En ce sens, RDF n'est pas un enfant de XML Schema mais plutôt un confrère, et il s'accompagne d'une spécification d'écriture de modèle qui lui est propre : RDF Schema. C'est ce qui va rendre son étude intéressante car, tout en étudiant les concepts liés à la mise en œuvre de métadonnées, nous allons voir un cas pratique où ni les DTD, ni XML Schema n'est utilisé pour modéliser des documents XML.

> VOCABULAIRE **Web sémantique**
>
> L'expression Web sémantique désigne l'ensemble des techniques nécessaires pour que la recherche d'information sur le Web soit rendue plus performante. Aujourd'hui, les mots des pages HTML sont indexés par les moteurs de recherche sans distinction de langue, terminologie, sens et critères de classification. Aussi, quand on cherche « cheval », d'une part on ne trouve pas « chevaux » mais, de plus, le système retourne sans distinction toutes les pages de toutes les utilisations possibles du mot cheval (cheval, « être à cheval sur », cheval d'arçon, fièvre de cheval, cheval de Troie, cheval fiscal…). Pour lutter contre ce bruit, il faut que les contenus des pages Web soient, en tout ou partie, classés, mis dans des rubriques thématiques et que la langue des vocabulaires qui s'y trouve utilisés soit elle-même reconnue.

Bien que sa première cible soit le Web sémantique, et donc HTML, RDF permet d'exprimer des ensembles de métadonnées en XML indépendamment du format intrinsèque de la ressource décrite. Il n'est même pas nécessaire qu'elle soit accessible informatiquement.

Au-delà de la description des ressources, RDF établit des relations entre leurs propriétés. Un modèle RDF peut, sur ce plan, être comparé à un diagramme de classes.

RDF respecte la liberté d'identification : chaque communauté culturelle, professionnelle, linguistique a ses propres mots-clés pour cataloguer l'information. Les mêmes mots sont importants dans certains contextes et vides de sens dans d'autres : le numéro ISBN d'un ouvrage concerne principalement les acteurs de la chaîne de diffusion, et plus rarement le lecteur pour lequel il se trouve n'être d'aucune utilité pour rechercher une information sur le Web ou dans les rayonnages d'un libraire. Le

numéro de sécurité sociale d'un individu n'est intéressant que si l'on sait qu'il s'agit d'un numéro de sécurité sociale, sinon c'est un nombre ordinaire. Un article de journal n'a de sens que si l'on connaît sa date de parution… Les exemples ne manquent pas montrant que les choses n'ont de sens que si on peut les placer par rapport à un référentiel de connaissances.

RDF ne donne pas de listes des noms à utiliser pour décrire une ressource mais spécifie les mécanismes pour les définir et les contrôler. De même que la recommandation XML Schema n'a jamais prétendu imposer un quelconque vocabulaire XML autre que le sien, RDF est un modèle permettant de définir un vocabulaire approprié aux caractéristiques d'une classe de ressources.

VOCABULAIRE Ressource au sens RDF

Pour RDF, une ressource est aussi bien un nœud d'un schéma RDF (il s'agit des éléments de base de type classe, propriété, type et domaine expliqués dans cette section), d'un fichier, d'un URI, d'une valeur exprimée à l'intérieur du contenu d'un élément ou d'une base de données. Toute ressource peut avoir des propriétés et être liée à une autre explicitement ou implicitement.

Concrètement, un modèle RDF traduit trois niveaux d'information :

- Une description concrète de ressources ; cela revient à exprimer des affirmations, par exemple « *la page Web http://www.cml.com/auto.html publiée par L. Nivert a été écrite par S. Avrane* », où l'adresse de la page Web et les deux noms propres sont des ressources.
- Une description logique des ressources et de leurs propriétés ; cela revient à exprimer des concepts tels que « une page Web », « un webmaster », « un auteur ».
- Un modèle pour établir des relations logiques et concrètes entre ces concepts et ressources. Par exemple : « Toute page Web est de la famille des pages HTML », « une page Web est éditée par un webmaster », « une page Web a un auteur », « le webmaster est un humain », « l'auteur est un humain », etc. Les relations expriment l'appartenance à une famille, une hiérarchie, un ordre…

RDF repose sur un modèle à deux dimensions : dans l'une, on classe l'information hiérarchiquement à la manière du système des poupées russes de XML et dans l'autre, on navigue dans l'information indépendamment de toute hiérarchie ; cette seconde dimension regroupe tous les liens.

La terminologie RDF repose sur quatre mots :

- La classe (`class`) qui représente un concept abstrait de classification des ressources ; par exemple, les classes des livres, humains, logiciels, auteurs, fruits...
- La propriété (`property`) qui définit une caractéristique particulière ; par exemple, les propriétés auteur, webmaster, éditeur, longueur, hauteur, couleur...

- Le type (`range`) qui spécifie la valeur autorisée d'une propriété ; par exemple, le nom d'un auteur doit être de type chaîne de caractères, un numéro de sécurité sociale doit être un nombre entier positif commençant par 1 ou 2.
- Le domaine (`domain`) qui précise les classes auxquelles s'applique une propriété ; par exemple, la propriété `auteur` s'applique aux classes des livres, thèses, articles de presse, chansons...

Ce modèle permet de créer les métadonnées indépendamment de la création physique des ressources alors que dans le monde issu de la programmation orientée objet, un objet doit d'abord être créé par instanciation d'un modèle avant qu'on puisse lui attribuer des propriétés par ailleurs limitées à la seule classe dont il est l'héritier. La figure 11-3 illustre ce modèle.

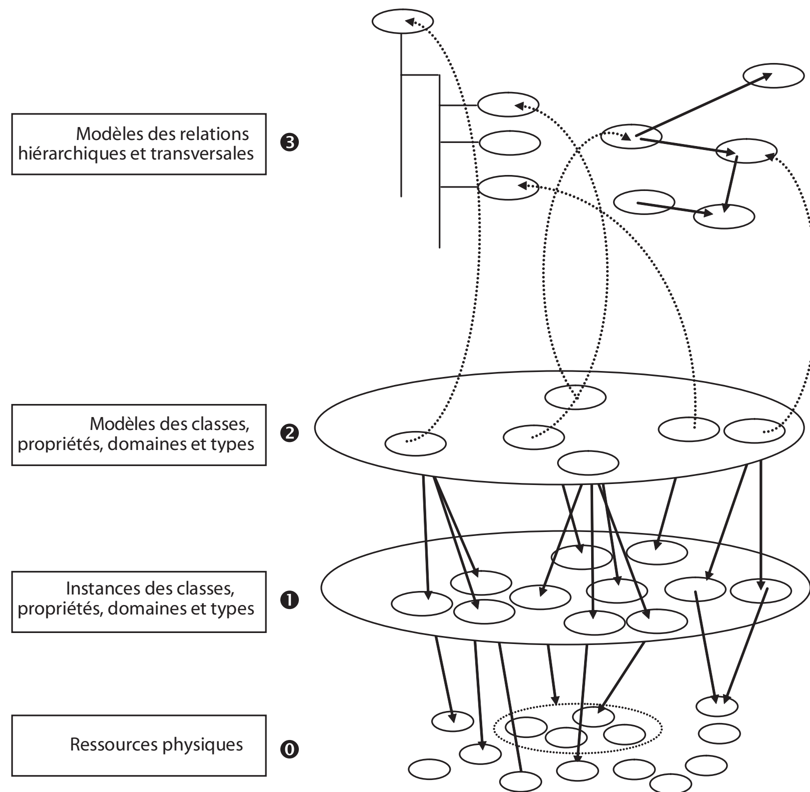


Figure 11–3
Les différents étages de la modélisation RDF

Des relations hiérarchiques sont établies à l'intérieur des objets de base : les classes, les propriétés, les domaines et les types peuvent être hiérarchisés tandis que des liens sont établis entre n'importe lequel de ces objets et même au-delà : les liens unissent les ressources RDF, quelles qu'elles soient.

En RDF, une métadonnée de base est un triplet composé d'une ressource, d'une propriété et de sa valeur. La figure 11-4 représente ce modèle on ne peut plus générique.

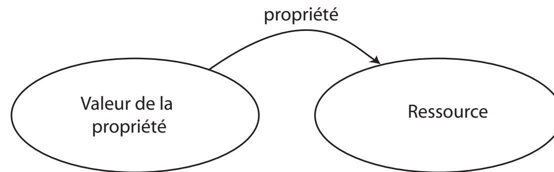


Figure 11–4
La relation élémentaire de RDF

Notez que, puisqu'en RDF les propriétés sont des objets à part entière, le modèle de base représenté à la figure 11-4 indique qu'une relation est établie, *via* la propriété établie entre une ressource et une valeur ; la valeur et la propriété étant mises en rapport avec une ressource et non pas l'inverse. Si RDF avait mis en œuvre la forme habituelle de la modélisation de type objet, le schéma aurait été inversé : la ressource aurait une propriété intrinsèque, ayant elle-même une valeur en propre qui lui aurait été attribuée lors de l'instanciation de l'objet. Dans le monde classique des objets, on ne peut accéder à une propriété qu'en passant par un objet. La représentation d'un tel modèle aurait dès lors été semblable à celle de la figure 11-5.

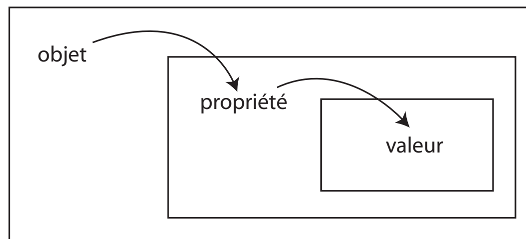


Figure 11–5
La relation classique entre un objet et ses propriétés

Commençons par un exemple de description d'une ressource :

```
<rdf:RDF>
 <rdf:Description about="http://www.poesie.com/illuminations">
  <ncn:Auteur>B. Estrade</ncn:Author>
 </rdf:Description>
</rdf:RDF>
```

On y exprime la relation entre la ressource http://www.poesie.com/illuminations, la propriété `ncn:Auteur` et la valeur de cette propriété `B. Estrade` exprimée en toutes lettres. Écrite ainsi, on a là un objet classique (la ressource) qui a une propriété, qui a elle-même une valeur.

Cependant, on aurait également pu écrire cette description sous la forme :

```
<rdf:RDF>
 <rdf:Description about="http://www.poesie.com/illuminations">
  <ncn:Auteur rdf:ressource=
             "http://www.poesie.com/auteurs.html#estrade"/>
 </rdf:Description>
</rdf:RDF>
```

Dans cet exemple, la valeur de la propriété est mentionnée par une URL qui renvoie probablement sur une description plus complète de l'auteur `B. Estrade`. Remarquez simplement que l'élément `ncn:Auteur` se comporte alors comme un élément vide : la valeur d'un élément est soit écrite entre les balises de début et de fin d'élément, soit *via* un attribut `href`, les deux formes s'excluant mutuellement.

L'exemple que nous voyons ici fait partie de l'application concrète de RDF. En résumé, ce document XML est une instance d'un modèle RDF Schema. On remarquera qu'il mêle deux vocabulaires : l'un issu de RDF reconnaissable à son préfixe `rdf` et l'autre issu d'un autre espace de noms et préfixé par `ncn`.

De la même manière que XML respecte une syntaxe formelle, le vocabulaire des éléments de base de RDF est défini formellement par une notation BNF. Ni les DTD ni les schémas XML ne sont utilisés : ils ne permettent pas d'exprimer certaines contraintes dont RDF a besoin.

Nous donnons ci-après la définition simplifiée de RDF.

```
[1] RDF ::= ['<rdf:RDF>'] description* ['</rdf:RDF>']
[2] description ::= '<rdf:Description' idAboutAttr? '>' property* '
    ➥ </rdf:Description>'
[3] idAboutAttr ::= idAttr | aboutAttr
[4] aboutAttr  ::= 'about="' URI-reference '"'
[5] idAttr   ::= 'ID="' IDsymbol '"'
[6] property ::= '<' propName '>' value '</' propName '>' | '<'
                 ➥ propName ressource '/>'
[7] propName ::= Qname
[8] value ::= description | string
[9] ressource   ::= 'ressource="' URI-reference '"'
[10] Qname ::= [ NSprefix ':' ] name
[11] URI-reference ::= string, interpreted per [URI]
[12] IDsymbol ::= (any legal XML name symbol)
[13] name  ::= (any legal XML name symbol)
[14] NSprefix ::= (any legal XML namespace prefix)
[15] string   ::= (any XML text, with "<", ">", and "&" escaped)
```

On y retrouve facilement les définitions des mots-clés utilisés dans notre exemple : les balises `<rdf:RDF>`, `<rdf:Description>` (règles [1] et [2]), les attributs `about` et `ressource` (règles [4] et [9]) et les règles qui autorisent l'utilisation de noms d'éléments provenant d'autres schémas (règles [6], [7], [10] et [13]) tel `ncn:Auteur` qui est un nom qualifié autorisé par la règle [6].

En XML Schema, la règle [6] aurait pu être déclarée ainsi :

```
<any namespace="##other" processContents="strict" minOccurs="0"
     maxOccurs="unbounded"/>
```

Cela ne serait pas du tout revenu au même :

- L'élément autorisé par RDF a un type de contenu bien imposé : il s'agit soit d'une description RDF, soit d'une chaîne de caractères.
- La règle 6 présente un choix entre un élément ayant un contenu et un élément vide. Dans le cas de l'élément vide, la règle [6] impose l'usage de l'attribut `href`. On aurait également pu exprimer un tel choix en XML Schema. Cela eût été absurde d'exprimer un choix en « n'importe quel élément » et un élément en particulier !

On constate ici sur un cas simple que XML Schema ne permettrait pas d'exprimer les contraintes dont les documents RDF ont besoin et que le recours à la notation BNF est ici justifié.

Comme nous venons de le voir, la description d'une ressource est simple. La véritable puissance de RDF se situe au-dessus, dans sa capacité à modéliser des hiérarchies de concepts et les liens entre les ressources comme cela est illustré à la figure 11-3, figure dans laquelle ce que nous venons de présenter est l'étage inférieur (repère ❶).

Parmi le vocabulaire autorisé pour décrire les propriétés des ressources, nous avons vu que RDF autorisait l'usage de n'importe quel vocabulaire de n'importe quel espace de noms. RDF bénéficie de cette possibilité et en profite pour définir un ensemble d'éléments d'usage commun.

Éléments complémentaires de RDF

Dans cette section, nous présentons quelques éléments RDF utilisés dans les descriptions de ressources. Ceux préfixés par `rdf` appartiennent à l'espace de noms des éléments de base et ceux préfixés par `rdfs` relèvent du vocabulaire complémentaire.

> DÉTAIL TECHNIQUE **Espaces de noms de RDF**
>
> RDF est défini à travers deux espaces de noms : l'un forme le noyau de RDF et l'autre est composé d'éléments et attributs complémentaires. Les espaces de noms correspondants sont respectivement http://www.w3.org/1999/02/22-rdf-syntax-ns# (le préfixe associé le plus souvent est `rdf`) et http://www.w3.org/2000/01/rdf-schema# (le préfixe le plus souvent associé est `rdfs`).

Pour illustrer le propos, nous nous appuyons sur un exemple concret :

```
<rdf:RDF xml:lang="en"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<rdf:Description ID="Vehicles">
 <rdf:type ressource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf
    ressource="http://www.w3.org/2000/01/rdf-schema#Ressource"/>
 <rdf:Category ressource="#vehiclesTypes">
</rdf:Description>
<rdf:Description ID="Constructeur">
 <rdf:type ressource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf ressource="http://www.w3.org/2000/01/rdf-
schema#Ressource"/>
</rdf:Description>
<rdf:Description ID="monoSpace">
 <rdf:type ressource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf ressource="#Vehicles"/>
 <rdfs:subClassOf ressource="#Constructeur"/>
</rdf:Description>
<rdf:Description ID="4x4">
 <rdf:type ressource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf ressource="#Vehicles"/>
 <rdfs:subClassOf ressource="#Constructeur"/>
</rdf:Description>
<rdf:Bag ID="vehiclesTypes">
 <rdf:li ressource="#monoSpace ">
 <rdf:li ressource="#4x4">
</rdf:Bag>

<rdf:Description ID="Propriétaire">
 <rdf:type
    ressource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
 <rdfs:domain rdf:ressource="#Vehicles"/>
 <rdf:Category ressource="#PropritairesTypes">
</rdf:Description>
<rdf:Alternative ID="PropriétairesTypes">
 <rdf:Description ID="personnePhysique">
  <rdf:type
     ressource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
  <rdfs:subPropertyOf rdf:ressource="#Propriétaire"/>
 </rdf:Description>
 <rdf:Description ID="personneMorale">
```

```
  <rdf:type
     ressource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
  <rdfs:subPropertyOf rdf:ressource="#Propriétaire"/>
  <rdfs:range
     rdf:ressource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
 </rdf:Description>
</rdf:Alternative>
</rdf:RDF>
```

Ce document RDF est un schéma au sens RDF, à savoir qu'il s'agit d'un modèle de métadonnées. Il fait intervenir, en plus de l'élément `rdf:Description` et de l'attribut `rdf:ressource` que nous avons déjà vus, les éléments et attributs `rdf:type`, `ID`, `rdfs:domain`, `rdfs:range`, `rdf:Property`, `rdfs:subClassOf` et `rdfs:subPropertyOf`.

Ce schéma RDF décrit des classes, des propriétés et des relations entre des ressources. Contrairement au premier exemple où l'on appliquait directement des valeurs de propriétés à des ressources concrètes via l'élément `rdf:Description`, il nous sert cette fois à créer de nouvelles classes de ressources. En particulier, on notera que l'élément `rdf:Description` sert ici à définir des ressources qui sont des nœuds intrinsèques au schéma. Finalement, on peut dire que ce schéma est un système de nœuds dont certains sont des classes et les autres des propriétés : tous peuvent cependant être utilisés comme ressources.

Notre exemple crée deux classes génériques, `Vehicles` et `Constructeur`. La classe `Vehicles` est composée de deux sous-classes qui sont les classes `monoSpace` et `4×4`. L'élément `rdf:Bag` permet de définir un groupe de classes ; ici, il est utilisé pour définir les sous-classes constitutives du type `VehiclesTypes`. Les classes `monoSpace` et `4×4` sont toutes les deux des sous-classes des deux classes principales `Vehicles` et `Constructeurs`. La propriété `Propriétaire` est rattachée à la classe `Vehicles` et les propriétaires peuvent être des personnes morales ou physiques.

REMARQUE Terminologie

Nous ramenons la terminologie générale de RDF à un vocabulaire proche de celui utilisé globalement dans cet ouvrage afin de rendre possible les comparaisons entre les différents modèles exposés, quitte à être réducteurs par rapport aux concepts réels décrits par la recommandation RDF.

La suite de cette section est consacrée à la description des éléments RDF utilisés dans cet exemple.

rdf:type

Cet élément indique le type de la ressource, et notamment s'il s'agit d'une classe ou d'une propriété. Les nœuds qui composent un document RDF sont de deux types : les ressources et les propriétés. Pour déclarer une ressource comme étant de type classe, il faut écrire :

```
<rdf:type ressource="http://www.w3.org/2000/01/rdf-schema#Class"/>
```

Et pour la déclarer comme propriété :

```
<rdf:type
   ressource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
```

Quand une ressource est déclarée comme classe, elle hérite de toutes les propriétés définies pour les membres de cette classe. Comme le montre notre exemple, une ressource peut être une instance de plus d'une classe (c'est le cas de `monoSpace` et `4×4`).

rdfs:subClassOf

Comme son nom l'indique, cet élément établit une relation de hiérarchie entre deux classes. Dans notre exemple, la relation peut être multivaluée : une classe peut avoir plusieurs généralisations. Ce point est fondamental afin que l'information puisse naviguer dans plusieurs directions.

> **VOCABULAIRE Hiérarchie de classe**
>
> Une relation de type hiérarchique entre classes est simplement une relation de type parent-enfant. Une classe dérivée d'une autre peut être considérée comme étant son enfant. Toutefois, une expression encore plus juste devrait être « relation de type spécialisation-généralisation » : elle indique qu'une classe est simplement plus générale qu'une autre, cette dernière étant qualifiée de spécialisée.

rdfs:subPropertyOf

L'élément `subPropertyOf` est utilisé quand une propriété est restrictive par rapport à une autre ou plusieurs. Les propriétés peuvent, par ce biais, être hiérarchisées. Dans l'exemple, on met en place un modèle permettant de décrire des liens familiaux. On écrit le fait que les propriétés `Père` et `Mère` sont des spécialisations de la propriété `Parents`, elle-même étant une spécialisation de la propriété `Ancêtres`.

```
<rdf:RDF xml:lang="en"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<rdf:Description ID="Ancêtres">
```

```
 <rdf:type
    ressource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Description>
<rdf:Description ID="Parents">
 <rdf:type
    ressource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
 <rdfs:subPropertyOf rdf:ressource="#Ancêtres"/>
</rdf:Description>
<rdf:Description ID="Père">
 <rdf:type
    ressource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
 <rdfs:subPropertyOf rdf:ressource="#Parents"/>
</rdf:Description>
<rdf:Description ID="Mère">
 <rdf:type
    ressource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
 <rdfs:subPropertyOf rdf:ressource="#Parents"/>
 </rdf:Description>
</rdf:RDF>
```

Ces liens expriment une hiérarchie de propriétés. En transposant les noms des propriétés en noms d'éléments XML, la logique exprimée par ces liens prendrait la forme suivante :

```
<Ancêtres ressource="…">
 <Parents ressource="…">
  <Père ressource="…"/>
  <Mère ressource="…"/>
 </Parents>
</Ancêtres>
```

On pourrait imaginer d'en écrire le schéma XML correspondant mais on rencontrerait un problème : à la différence d'un schéma XML, un schéma RDF permet d'écrire un document RDF réduit à l'élément `père` sans pour autant perdre la logique Ancêtres/Parents/Père. En d'autres termes, un schéma RDF définit la sémantique des relations entre les ressources que les documents RDF n'ont ensuite plus besoin d'exposer.

REMARQUE Récursivité

Pour les deux éléments précédents, la récursivité est interdite, qu'elle soit directe ou indirecte (une sous-classe ne peut pas se trouver être une sous-classe d'elle-même et il en va de même avec les sous-propriétés).

rdfs:seeAlso et rdfs:isDefinedBy

L'élément seeAlso spécifie un lien de type index. L'élément isDefinedBy représente une sous-propriété de seeAlso. Dans les deux cas, la valeur de ces éléments est une ressource.

rdfs:domain et rdfs:range

L'élément rdfs:domain spécifie la classe à laquelle s'applique une propriété (voir figure 11-4). Par exemple, on utilise rdfs:domain pour indiquer que la propriété auteur s'applique aux objets (ressources) de la classe Livre. Une propriété peut avoir comme domaine zéro, une ou plusieurs classe(s). S'il n'y a pas de domaine, la propriété peut être utilisée dans n'importe quelle ressource, et quand un domaine est défini, la propriété ne peut alors être utilisée que pour les objets (ressources) des classes correspondantes.

L'élément rdfs:range permet de spécifier la valeur d'une propriété en indiquant la classe à laquelle cette valeur doit appartenir. Par rapport à ce qui ce fait habituellement en XML pour définir un type de contenu, RDF utilise ici un mécanisme original. Par exemple, rdfs:range permet d'imposer qu'un nom d'auteur soit une ressource appartenant à la classe des personnes. La valeur de rdfs:range ne doit correspondre qu'à une et une seule classe (si on souhaite qu'elle s'applique à plusieurs classes, il suffit de passer par une super-classe).

Le fragment ci-après est un exemple d'utilisation des éléments rdfs:range et rdfs:domain. On y spécifie que la propriété millésime de la classe des entiers de XML Schema (xs:interger) s'appliquera aux objets de la classe vehicles. Quant à elle, la classe vehicles est une sous-classe de la classe mère de toutes les classes : rdf-schema#Ressource.

```
<rdf:RDF xml:lang="en"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
<rdfs:Class rdf:ID="vehicles">
 <rdfs:subClassOf
    rdf:ressource="http://www.w3.org/2000/01/rdf-schema#Ressource"/>
</rdfs:Class>
<rdf:Property ID="millésime">
 <rdfs:range rdf:ressource="xs:integer"/>
 <rdfs:domain rdf:ressource="#vehicles"/>
</rdf:Property>
</rdf:RDF>
```

Métadonnées relatives au contenu du document

Depuis le début de ce chapitre, nous avons étudié les métadonnées qui s'appliquent à la totalité d'un document XML, voire d'une collection de documents XML ou ressources. En quelque sorte, ces métadonnées figuraient à l'extérieur des documents eux-mêmes.

Nous allons désormais plonger au cœur des documents XML et étudier, à partir de cette section, les métadonnées qui peuvent y être utilisées.

Il n'existe, dans un document, que deux moyens pour exprimer des métadonnées :

- les attributs
- les éléments.

Métadonnées transmises par un attribut

L'attribut qui porte la métadonnée peut être de deux natures différentes :

- Il est public et appartient à un espace de nom spécialisé dans la transmission de métadonnées.
- Il est privé et appartient à l'application.

Nous allons dans cette section prendre pour exemple les cas mis en œuvre par SOAP ; il s'agit d'un modèle conçu pour transmettre des données, et en particulier les métadonnées qui les accompagnent.

Une première possibilité est de référencer le schéma XML qui contient la définition du type de la donnée transportée. Cette méthode n'est applicable que dans le cas de documents XML bien formés, qui ne se sont pas vu attribuer de schéma XML global.

```
<t:identification xmlns:t="http://www.carBuy.org/schema.xsd">05-1997
</t:identification>
```

Le fragment précédent indique que la valeur `05-1997` transportée dans l'élément `identification` est du type défini pour l'élément `identification` dans le schéma dont l'URI est http://www.carBuy.org/schema.xsd.

Une deuxième technique consiste à utiliser l'attribut flottant `xsi:type` et de faire, comme dans l'exemple suivant, un typage dynamique des éléments :

```
<identification xsi:type="xsd:int"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">51997
</identification>
```

Dans cet exemple, on assigne le type entier de XML Schema à l'élément `identification`.

Métadonnées transmises par une structure d'éléments

Enfin, une troisième technique consiste à utiliser une structure plus complexe. SOAP, par exemple, autorise à déclarer dynamiquement le modèle de contenu d'un élément au travers de deux mécanismes originaux : les accessoires polymorphes et les tableaux d'éléments.

> **INFORMATION SOAP**
>
> SOAP est un protocole d'échange de données adapté à Internet et aux besoins des services Web.

Les tableaux d'éléments sont définis au moyen du type `soap-enc:Array` :

```
<element name="myFavoriteNumbers" type="soap-enc:Array"/> ← ⓿
```

Dans cet exemple, l'élément `myFavoriteNumbers` est déclaré comme étant de type tableau (repère ⓿).

L'étape suivante consiste à indiquer le type des éléments constitutifs du tableau. Cela se fait, lors de l'utilisation du tableau, en utilisant l'attribut `soap-enc:arrayType` (repère ❶) :

```
<myFavoriteNumbers soap-enc:arrayType="xsd:int[2]"> ← ❶
 <number>3</number> ← ❷
 <number>4</number> ← ❸
</myFavoriteNumbers>
```

Le type choisi est ici `xsd:int`, ce qui signifie que c'est le type entier de XML Schema qui est retenu (repères ❷ et ❸).

Une autre possibilité autorisée est d'utiliser directement l'élément `Array`, comme ceci :

```
<enc:Array xmlns:enc="http://www.w3.org/2001/12/soap-encoding"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  enc:ArrayType="xs:int[2]" >
<enc:int>3</enc:int>
<enc:int>4</enc:int>
</enc:Array>
```

Remarquez que, dans ce fragment, on a déclaré l'espace de noms de l'élément `Array`. La dimension du tableau a été spécifiée au moyen de l'attribut `ArrayType`, et elle est ici de 2. Toutefois, il serait trompeur de croire que cet attribut impose également le type des données transmises : leur véritable type est spécifié par l'utilisation directe du nom du type comme nom d'élément, à savoir `enc:int`. Dans SOAP, chaque type prédéfini de XML Schema se voit associé à un élément qui porte son nom. Cela est l'un des mécanismes particuliers de SOAP. Il comporte une contrainte toutefois : le type utilisé au niveau des éléments du tableau doit être une dérivation valide de celui déclaré au niveau de l'attribut `ArrayType`.

Aussi, en utilisant le père de tous les types dans l'attribut `enc:ArrayType`, c'est-à-dire le type `ur` (repère ❿), on peut créer des tableaux contenant des données de n'importe quel type (repères ⓿, ❶, ❷ et ❸). L'exemple suivant fournit deux cas d'application. Dans l'un, un élément intermédiaire `thing` est utilisé pour porter les données, et dans l'autre elles sont portées par des éléments SOAP autodescripteurs de leur type de contenu.

```
<soap-enc:Array soap-enc:arrayType="soap-enc:ur-type[4]"> ← ❿
 <thing xsi:type="xsd:int">12345</thing> ← ⓿
 <thing xsi:type="xsd:decimal">6,789</thing> ← ❶
 <thing xsi:type="xsd:string"> ← ❷
 Of Mans First Disobedience, and the Fruit
 Of that Forbidden Tree, whose mortal tast
 Brought Death into the World, and all our woe,
 </thing>
 <thing xsi:type="xsd:uriReference"> ← ❸
  http://www.dartmouth.edu/~milton/reading_room/
 </thing>
</soap-enc:Array>
<soap-enc:Array soap-enc:arrayType="soap-enc:ur-type[4]"> ← ❿
 <soap-enc:int>12345</soap-enc:int> ← ⓿
 <soap-enc:decimal>6,789</soap-enc:decimal> ← ❶
 <xsd:string> ← ❷
 Of Mans First Disobedience, and the Fruit
 Of that Forbidden Tree, whose mortal tast
 Brought Death into the World, and all our woe,
 </xsd:string>
 <soap-enc:uriReference> ← ❸
  http://www.dartmouth.edu/~milton/reading_room/
 </soap-enc:uriReference >
</soap-enc:Array>
```

Nous venons d'étudier un premier type de tableaux, celui où les données transmises sont de type simple. Nous allons désormais voir comment transmettre des tableaux de données complexes.

À la différence des tableaux de type `Array`, ceux de type `Struct` permettent de transporter des structures complexes : chaque élément du tableau ne contient plus directement la valeur à transmettre, mais une série de balises emboîtées les unes dans les autres. La syntaxe définie pour spécifier le nom de l'élément composé qui servira d'item au tableau est très proche de ce que nous venons de voir. Comme le montre l'exemple suivant, le nom qualifié de l'élément complexe utilisé dans le tableau est déclaré en lieu et place du nom du type simple dans la valeur de l'attribut `soap-enc:arrayType`. Le repère ❿ indique la ligne utilisée pour déclarer l'espace de noms de l'élément `Order`, et le repère ⓫ celle où est déclaré le tableau composé des éléments de type `Order`. Les repères ⓿ et ❶ indiquent les endroits où sont concrètement utilisés les éléments du tableau ainsi définis.

```
<soap-enc:Array xmlns:enc="http://www.w3.org/2001/12/soap-encoding"
   xmlns:xyz=http://example.org/2001/06/Orders ← ❿
    soap-enc:arrayType="xyz:Order[2]"> ← ⓫
 <Order> ← ⓿
  <Product>Pomme</Product>
  <Price>1,56</Price>
 </Order>
 <Order> ← ❶
  <Product>Pêche</Product>
  <Price>1,48</Price>
 </Order>
</soap-enc:Array>
```

La technique du référencement de contenus externes par le mécanisme des `id/href` détaillé plus avant permet de construire des tableaux de tableaux. L'exemple suivant est un tableau composé de deux tableaux (repères ❿, ⓿ et ❶). Les sous-tableaux sont eux-mêmes constitués respectivement de trois (repère ❷) et deux (repère ❸) chaînes de caractères.

```
<soap-enc:Array soap-enc:arrayType="xsd:string[][2]"> ← ❿
 <item href="#array-1"/> ← ⓿
 <item href="#array-2"/> ← ❶
</soap-enc:Array>
<soap-enc:Array id="array-1" soap-enc:arrayType="xsd:string[3]"> ← ❷
 <item>r1c1</item>
 <item>r1c2</item>
 <item>r1c3</item>
</soap-enc:Array>
<soap-enc:Array id="array-2" soap-enc:arrayType="xsd:string[2]"> ← ❸
 <item>r2c1</item>
 <item>r2c2</item>
</soap-enc:Array>
```

Les tableaux peuvent être *multidimensionnels*. Pour cela, la valeur entre crochets devient un couple de valeurs : la première donne le nombre de sous-tableaux et la seconde le nombre d'items de chacun d'eux. Dans l'exemple suivant, on spécifie que les deux sous-tableaux sont constitués de trois items :

```
<soap-enc:Array soap-enc:arrayType="xsd:string[2,3]">
<!-- première série -->
 <item>r1c1</item>
 <item>r1c2</item>
 <item>r1c3</item>
<!-- deuxième série -->
 <item>r2c1</item>
 <item>r2c2</item>
 <item>r2c3</item>
</soap-enc:Array>
```

Même si SOAP contient d'autres mécanismes qui permettent, entre autres, de ne transmettre que quelques valeurs d'un tableau, nous en avons vu ici l'essentiel, à savoir un mécanisme sophistiqué pour déclarer dynamiquement des types dans le corps même du document XML.

En résumé

Dans ce chapitre, nous avons passé en revue plusieurs techniques de transmission des métadonnées avec les documents XML.

On peut y procéder au travers des schémas XML quand ils existent et sont accessibles ou, de manière peu exploitable, en produisant le PSVI du document XML avant de le transmettre.

Les métadonnées peuvent concerner une donnée élémentaire, et servent alors à en exprimer le type et potentiellement leur sémantique, ou concerner l'entité document et représenter alors ses données de gestion.

Des jeux tout faits de métadonnées sont disponibles : c'est le cas du Dublin Core, un ensemble prédéfini d'éléments valables pour tous types de documents textuels et de la S1000D, mais, dans ce dernier cas, les métadonnées sont spécifiques à un métier bien particulier. Toutefois, des entreprises se sont déjà inspirées de ces modèles pour développer leurs propres jeux de métadonnées.

Enfin, nous avons vu des modèles qui servent à décrire des modèles plutôt sophistiqués de métadonnées, cela indépendamment des documents XML eux-mêmes. Le

modèle RDF permet via les métadonnées de décrire des bases de connaissances et d'établir entre les ressources de ces bases des relations typées.

Quant à SOAP, nous avons utilisé ce standard pour montrer le mécanisme original qui est mis en œuvre afin de transmettre dynamiquement des informations de typage, tant sur des structures simples que complexes, en l'occurrence des tableaux de données.

La gestion des métadonnées nous amène naturellement vers la gestion des liens, qui fait l'objet du chapitre suivant. En effet, comme nous l'avons vu avec RDF, gérer des métadonnées sans gérer les relations qui les rapprochent serait comme donner un coup d'épée dans l'eau. Ainsi, nous allons voir au chapitre suivant les mécanismes susceptibles d'être mis en œuvre pour gérer des liens, du plus simple au plus compliqué.

12

Modèles pour la gestion des liens

Le modèle XML est particulièrement propice à la pose de liens. Comme nous allons le voir dans ce chapitre, ce sujet est particulièrement critique et exacerbé :

- Critique, car a priori le moindre des documents XML contient des liens. Il peut évidemment se produire des cas où des documents XML ne contiennent pas de liens, mais c'est très rare.
- Exacerbé, car il peut arriver que des documents XML n'aient physiquement aucun contenu ; ce dernier est alors entièrement stocké à l'extérieur du document d'où il est référencé par des liens. Dans certains cas également, des documents XML peuvent avoir pour seule vocation de définir des faisceaux de liens ; nous en avons eu un aperçu au chapitre précédent avec RDF. On y a même considéré la possibilité de typer un lien et d'obtenir ainsi une relation.

On dit de ces cas extrêmes que ce sont des documents massivement hyperliés. Il ne s'agit pas de cas exceptionnels : un catalogue de composants d'un système complexe peut contenir jusqu'à 100 000 liens, une encyclopédie entre 400 000 et 500 000, etc.

Le problème des liens est général au monde des documents et s'étend donc à celui de la donnée.

Nous allons étudier dans ce chapitre les différents cas de liens et les solutions à apporter à cette question.

Les différents types de liens

VOCABULAIRE Lien UML vs lien XML

Dans UML, le terme de lien désigne une relation faite entre objets au moyen de leur référence, laquelle permet de définir le rôle desdits objets. Dans XML, le lien désigne le mécanisme qui permet de passer d'un élément à un autre sans transiter par la hiérarchie parent-enfant propre à XML. Ainsi, le lien permet de passer d'un élément A à un élément B de façon totalement indépendante de la structure des documents dans lesquels se trouvent les éléments A et B. On appelle cette forme de déplacement à l'intérieur d'un document XML « navigation transversale non-linéaire » en opposition à « la navigation linéaire » qui, dans un document XML, représente des déplacements qui suivent l'arborescence XML du document.

Depuis le lien simple reliant deux éléments XML jusqu'au lien sophistiqué exprimé sous la forme d'une expression calculée, XML offre une palette de possibilités que nous allons décrire dans cette section. Nous avons déjà eu l'occasion, dans les chapitres précédents, d'évoquer la question en présentant des liens de type RDF au chapitre 11.

En XML, les liens, comme les données, peuvent être typés et avoir leur propre modèle : en d'autres termes, les liens sont exprimés directement ou au travers de structures XML. Listons d'ores et déjà les catégories suivantes :

- Les liens de type document : renvois de tables des matières, figures, tableaux, index, bibliographies… Ils sont calculés.
- Les liens informatifs de navigation transversale : références croisées entre un texte et une illustration, un tableau, etc. Ils sont posés par les auteurs ou calculés à partir de règles métier.
- Les liens de navigation hiérarchique par la structure : liens entre chapitres, sections, etc. Ils sont définis implicitement par le schéma XML et sont calculés automatiquement. On peut naviguer de deux manières : l'axe parent-enfant (on saute de chapitre en section…) et l'axe des noms d'éléments (on saute de paragraphe en paragraphe, de titre en titre, etc.)
- Les liens volatils de navigation par le contenu : suite à une recherche de mots, on utilise par exemple la fonction « mot suivant ».
- Les liens de référencement, qui sont des pointeurs vers des objets stockés à l'extérieur du document (programmes, animations, séquences visuelles, etc.).
- Enfin, les liens d'inclusion de contenu : il s'agit d'appels pour aller chercher du contenu à l'extérieur du document.

Quel que soit le cas de figure, les liens définis sont soit logiques, soit physiques, à savoir :

- Un lien logique associe indirectement des ressources physiques ; généralement *via* des descriptions abstraites de ces ressources physiques.

- Un lien physique associe physiquement deux (ou plus) ressources physiques : une URL en est un exemple typique.

Toutes ces distinctions vont être présentées dans les prochaines sections.

Liens logiques et physiques

Liens physiques

Un lien physique associe deux ressources en indiquant dans l'une le chemin d'accès physique qui conduit à l'autre. Comme à la figure 12-1, les URL de HTML sont les meilleurs représentants de cette catégorie.

Ces liens posent le problème du déplacement d'une ressource : il faut dans ce cas changer « en dur » dans toutes les pages HTML l'ensemble des URL pointant vers la ressource déplacée.

Ce problème n'est même pas minimisé par l'utilisation d'URL relatives. En effet, l'intérêt des URL relatives est seulement de pouvoir déplacer en bloc un ensemble de pages HTML. Cependant, quand une seule est déplacée, le problème mentionné précédemment persiste.

VOCABULAIRE Ressource

On appelle ressource tout objet d'un système adressable au moyen d'une URL.

VOCABULAIRE URL

URL est l'acronyme de Uniform Resource Locator, en français *pointeur uniforme de ressources*. Il s'agit d'une forme de syntaxe donnant une adresse universelle à un objet informatique ; c'est le `http://www.xxx.yy/abc`... bien connu des internautes. Les URL ne se limitent toutefois pas au seul protocole HTTP et à Internet, et toutes les possibilités d'adressage sont définies par la RFC 1738 de décembre 1994 (les RFC sont des normes émises par l'IETF – Internet Engineering Task Force).

TECHNIQUE URL absolue et URL relative

Une URL absolue exprime l'adresse d'une ressource à partir de la racine de la machine sur laquelle elle se trouve. Par exemple, l'URL absolue du texte de l'IETF définissant les URL est `http://www.ietf.org/rfc/rfc1738.txt`.
A contrario, une URL relative est donnée par rapport à une autre. Cela est possible à condition que les deux ressources se trouvent sur la même machine. Par exemple, des URL relatives du même texte normatif pourraient être `../rfc1738.txt` ou `./rfc1738.txt`, ou encore `../../rfc1738.txt`.

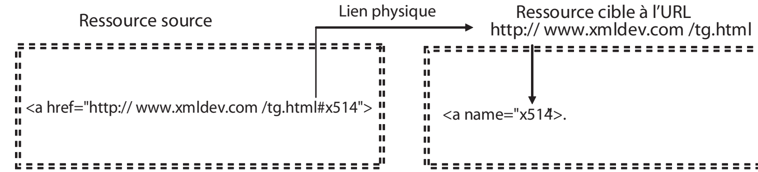


Figure 12–1
Schéma du principe de fonctionnement d'un lien physique

Liens logiques

Un lien est logique quand la cible est spécifiée au moyen d'identifiants la représentant logiquement, par exemple en utilisant des URI – Uniform Resource Identifier (en français, *identifiant uniforme de ressource*). Ces derniers doivent, à un moment donné, être transformés en identifiants physiques (en URL par exemple) pour que la ressource cible soit connue et atteinte. La figure 12-2 illustre ce cas de figure.

> **VOCABULAIRE URI**
>
> Un URI, ou identifiant uniforme de ressource, est un nom donné à une ressource. Un URI répond aux mêmes règles d'écriture qu'une URL mais avec cette différence qu'il n'est pas censé être le chemin d'accès réel de la ressource. Pour trouver la ressource physique, il faut établir une correspondance entre l'URI de la ressource et son URL.

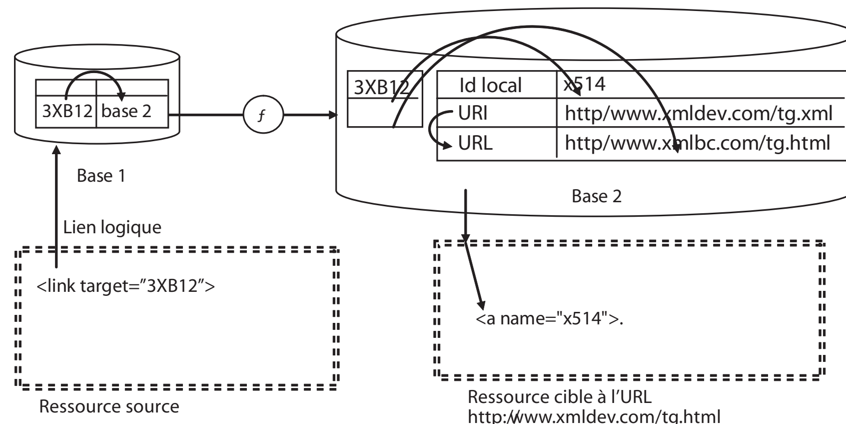


Figure 12–2
Schéma du principe de fonctionnement d'un lien logique

La caractéristique des liens logiques est qu'ils peuvent être définis à l'extérieur du document XML dans lequel on ne place plus que des indicateurs, qu'il s'agisse d'identifiants ou d'autres techniques de ciblage telles que des expressions Xpath.

Dans la figure 12-2, les identifiants logiques locaux sont les chaînes de caractères `3XB12` et `x514`. L'identifiant `3XB12` est logique : il ne correspond directement à aucune

ressource physique. Pour la connaître, il faut qu'un programme interprète cet identifiant logique en mettant en œuvre l'algorithme suivant :

- tout d'abord, savoir qu'il doit consulter la base 1 ;
- puis trouver la base cible associée à l'identifiant logique ;
- atteindre la base 2 au moyen d'une fonction *f* ;
- trouver dans la base 2 l'identifiant logique local cible ;
- trouver dans la base 2 l'URI (adresse logique) de la ressource cible (celui où se trouve l'identifiant logique cible) ;
- trouver dans la base 2 l'URL (adresse physique) correspondant à l'URI trouvé à l'étape précédente.

Dans la figure 12-2, l'interprétation de la cible logique se fait à l'extérieur du document, et on suppose que c'est dans une base de données. Il n'est toutefois pas interdit de mettre ces informations en en-tête de document.

L'utilisation de liens logiques présente de nombreux avantages. Cela augmente la flexibilité des systèmes contenant un grand nombre de documents XML, la transportabilité des documents et l'adaptation des liens à des contextes particuliers.

Dans la section suivante, nous présentons un exemple concret mettant en œuvre des liens logiques.

Exemple de mise en œuvre d'un lien logique

L'exemple que nous utilisons ici est un cas réel. Il s'agit d'un modèle élaboré pour gérer des liens à l'intérieur d'une publication résultant de l'assemblage de plusieurs dizaines de milliers de petits fragments dénommés modules. C'est une bonne illustration du concept présenté à la figure 12-2.

Dans le module suivant, le texte de l'élément `<p>` contient une référence `L1` à un élément `linkId` situé en début de module. Il a pour rôle de spécifier le lien de manière logique : la cible est l'élément porteur de l'identifiant `i5` dans le module dont l'identifiant logique est `e07652`.

```
<module>
<linkId id="L1">
  <target id="e07652" type="element">i5</target>
</linkId>
......
<p id="p1"> Ce paragraphe contient un lien vers un <link ref="L1">
élément</link> d'un autre document.</p>
</module>
```

Un autre document XML établit la correspondance entre l'identifiant logique de la cible et l'URL de la ressource cherchée. Dans notre exemple, cette ressource est le fichier `c:/doc/6733.xml` :

```
<docbase>
 <ressourceId id="L07652" type="entity">e07652</ressourceId>
 <ressource url="file://c:/doc/6733.xml" ref="L07652"/>
</docbase>
```

La situation peut être résumée au travers des tableaux 12-1 et 12-2 ci-après.

Tableau 12–1 Cheminement logique dans la source du lien

L1	Identifiant logique local.
↓	
e07652	Identifiant logique de la ressource cible.
I5	Identifiant physique de l'élément ciblé dans la ressource cible.

Tableau 12–2 Cheminement logique dans le fichier de correspondances

e07652	Identifiant logique de la ressource cible.
↓	
L07652	Identifiant logique interne.
↓	
c:/doc/6733.xml	Adresse physique de la ressource cible.

L'identifiant physique `i5` de l'élément XML aurait également pu être un identifiant logique. Son processus de transposition aurait alors suivi un schéma similaire à celui que nous venons de décrire. Cet identifiant aurait également pu être une expression Xpointer, recommandation du W3C qui permet de définir une cible au moyen d'une expression calculée.

Nous avons glissé dans le module XML de départ un identifiant `p1` sur l'élément `p`, pour indiquer que cet élément peut lui-même être une cible de lien. Le cas de deux modules XML disposant de liens logiques croisés l'un vers l'autre est illustré à la figure 12-3.

Dans cette figure sont représentés trois documents XML : deux sont des modules nommés `mod1` et `mod2`, et le troisième est une base de transposition qui spécifie les correspondances logique/physique.

Dans ce schéma, la base de transposition sert en quelque sorte de gare de triage des liens. Elle prend en charge la mise en relation des sources logiques avec leurs cibles physiques.

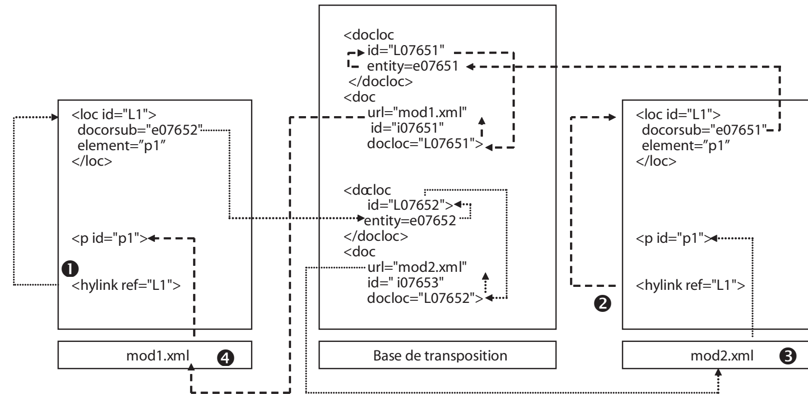


Figure 12–3
Exemple de liens croisés entre deux modules

On notera que, dans notre exemple, sources et cibles se sont vu attribuer les mêmes identifiants, à savoir `L1` pour les sources et p1 pour les cibles. Avec la technique que nous décrivons ici, cela n'est pas gênant.

Les points de départ des liens sont repérés par les marques ❶ et ❷. Leurs cibles sont les éléments repérés par les marques ❸ et ❹. Le tracé en pointillé fin indique le chemin suivi pour aller de ❶ à ❸, tandis que celui figurant en pointillé épais montre le chemin suivi pour aller de ❷ à ❹.

Tous les identifiants logiques des cibles utilisés dans les modules sont mis en correspondance avec une adresse physique dans la base de transposition. Toutefois le mécanisme, comme vous pouvez le voir en suivant les traits en pointillé, n'est pas direct et passe lui-même par un jeu de liens internes à la base de transposition : cela afin d'éviter qu'une même adresse physique soit répétée autant de fois qu'il y a de liens l'utilisant dans les documents.

Nous donnons ci-après la représentation concrète de ces trois documents XML :

Module `mod1.xml` :

```
<module>
 <avcorps>
  <linkId id="L1">
   <target id="e07652" type="element">i5</target>
  </linkId>
 </avcorps>
 <body>
  <p id="p1"> Ce paragraphe contient un lien vers un <link ref="L1">
élément</link> de document 2.</p>
 </body>
</module>
```

Module `mod2.xml` :

```
<module>
 <avcorps>
  <linkId id="L1">
   <target id="e07651" type="element">p1</target>
  </linkId>
 </avcorps>
 <body>
  <p id="i5"> Ce paragraphe contient un lien vers un <link ref="L1">
élément</link> de document 1.</p>
 </body>
</module>
```

Base de transposition :

```
<docbase>
  <ressourceId id="L07651" type="entity">e07651</ressourceId >
  <ressourceId id="L07652" type="entity">e07652</ressourceId >
  <ressource url="file://c:/doc/mod1.xml" ressourceId="L07651"/>
  <ressource url="file://c:/doc/mod2.xml" ressourceId="L07652"/>
</docbase>
```

Nous en profitons pour attirer votre attention sur la différence que nous avons notée dans les chapitres 2 et 3 entre les modèles conceptuels et les modèles physiques. Le modèle conceptuel indique la sémantique des liens qui sont mis en jeu dans le modèle, comme l'indique la figure 12-4 :

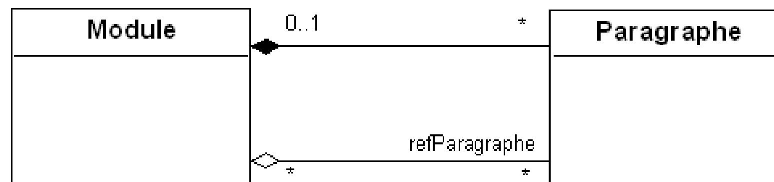


Figure 12–4 Exemple de modèle conceptuel représentant le lien entre les entités module et paragraphe

L'association de composition indique que le modèle physique XML utilisera vraisemblablement le mécanisme d'emboîtement des éléments XML, ce qui est le cas dans notre exemple. Rien n'est dit, en revanche, sur le mode d'implémentation physique de l'association de référencement de paragraphe.

On pourrait faire un modèle de classe de l'implémentation des liens de référence ; mais il se situerait *de facto* sur un autre plan conceptuel que le modèle représenté à la figure 12-4.

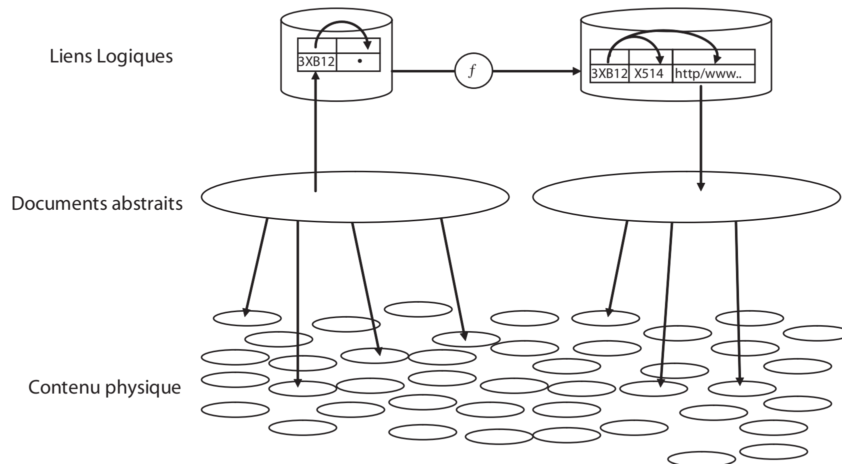
Cas extrême

Dans certains cas, la totalité du contenu d'un document est stockée à l'extérieur du document. Il n'est alors plus qu'un lieu de rassemblement des liens utilisés pour collecter l'information au moment de sa publication.

Un tel document n'est finalement constitué que de liens vers la matière stockée à l'extérieur : des mots, phrases, paragraphes, illustrations, icônes, etc.

Ces liens viennent se superposer aux autres et forment un ensemble que nous avons représenté à la figure 12-5. Il s'agit précisément des liens reliant la couche intitulée « documents abstraits » à la couche « contenu physique ».

Figure 12–5
Documents abstraits composés de liens logiques et de liens vers le contenu



Ces documents abstraits ont cet avantage, de toute évidence, qu'ils sont extrêmement souples, notamment en matière de mises à jour, de traductions, d'adaptations. Ils ne peuvent toutefois exister que dans un monde parfaitement modélisé et structuré.

Nous ne rentrerons pas ici dans le détail de tels cas d'utilisation extrêmes, dont les problèmes de prise en compte relèvent bien plus du domaine de la gestion que de celui de la mécanique des liens.

Liens simples de type ID/IDREF

Les liens simples de type ID/IDREF existent en XML 1.0 et perdurent en XML Schema simplement pour assurer la compatibilité ascendante de XML.

Le type ID (comme IDentifiant) attribue à un élément un identifiant unique. Un exemple déjà utilisé dans la section précédente est `<p id="p1">`. L'identifiant `p1` est ici attribué à l'élément p au moyen de l'attribut `id`. Ni la valeur `p1` ni le nom de l'attribut `id` n'est imposé(e), bien évidemment. Le type ID est dévolu à l'attribut `id` lors de sa définition dans la DTD.

Le type IDREF (comme REFérence d'IDentifiant) permet de référencer un identifiant unique ; par exemple en écrivant `<ref idref="p1"/>`. Là encore, ni le nom de l'attribut `idref` ni sa valeur n'est imposé(e). Le type `IDREF` est dévolu à l'attribut `idref` lors de sa définition dans la DTD.

Remarquez que le mécanisme des ID/IDREF ne permet pas autre chose que de faire des liens simples, non typés et sans définition particulière de comportement. Une limitation majeure des types ID et IDREF est que les sources et cibles sont obligatoirement confinées à l'intérieur du même document XML :

- La valeur de l'attribut de type ID doit être unique à l'intérieur du document qui la contient, et c'est une unicité totale : elle ne peut être confinée à une partie seulement du document.
- La valeur de l'attribut de type IDREF doit être un identifiant unique du document dans lequel elle se trouve.

Les autres limitations sont les suivantes :

- Pas de support des expressions Xpath.
- Les types ID/IDREF ne sont applicables qu'à des valeurs d'attributs et non au contenu des éléments.

En conclusion de ces limitations, on peut affirmer que le type ID/IDREF est notoirement insuffisant pour bâtir une quelconque stratégie de gestion des liens au sein d'un système d'information.

L'approche du W3C : XLink

Introduction

XLink est la recommandation du W3C qui spécifie un vocabulaire pour établir des liens entre des ressources. Ces liens peuvent être de type simple (point à point) ou étendus, c'est-à-dire utilisant des relations typées.

La recommandation date du 27 juin 2001 (http://www.w3.org/TR/2001/REC-xlink-20010627/). Elle n'a pas été modifiée à ce jour.

Nous allons présenter dans cette section les liens simples et étendus, et nous verrons un cas réel de mise en œuvre.

Les liens XLink sont définis au moyen d'attributs. Ces derniers doivent être déclarés dans les DTD et schémas XML pour être valides. Une fois les liens posés, leur sémantique est comprise nativement par les logiciels prenant en charge la recommandation XLink.

Il s'agit de liens de navigation à ne pas confondre avec des liens d'inclusion. Comme leur nom l'indique, les liens de navigation servent à naviguer de documents en documents, tandis que les liens d'inclusion servent à inclure logiquement des documents XML les uns dans les autres : en termes de modèles XML, la différence est de taille puisque, dans le premier cas, les schémas doivent seulement cohabiter, tandis que dans le second ils doivent être parfaitement compatibles.

Afin de rendre explicite la présentation de XLink qui suit, nous allons utiliser un exemple dans lequel interviendront des personnes et des liens entre ces personnes :

Liste des personnes :

- P1 : Mme GR,
- P2 : M. DH,
- P3 : M. JG.

Liens entre les objets :

- R1 : « est la mère de »,
- R2 : « est le père de »,
- R3 : « est la grand-mère de ».

À partir de ces données, on simulera les situations suivantes :

- P1(I1)-R1-P2(I2) : Madame GR est la mère de monsieur DH.
- P2(I2)-R2-P3(I3) : Monsieur DH est le père de monsieur JG.
- P1(I1)-R3-P3(I3) : Madame GR est la grand-mère de monsieur JG.

Dans une telle relation de parenté, les relations R1, R2 et R3 ne sont pas définies en dur. En revanche, une relation générale unissant deux êtres peut être représentée ensuite, cas par cas, en précisant le rôle des personnes dans la relation. On obtient ainsi le modèle exprimé par le tableau 12-3.

Tableau 12–3 Expression de relations de parenté

Personne	Rôle	f	Personne	Rôle
Madame GR	Mère	Relation de parenté	Monsieur DH	Fils
Monsieur DH	Père	Relation de parenté	Monsieur JG	Fils
Madame GR	Grand-mère	Relation de parenté	Monsieur JG	Petit-fils

Les liens établis sont alors bijectifs puisque le tableau inverse peut être déduit très facilement de ces relations.

Complétons le tableau 12-3 en attribuant des propriétés aux personnes ; en l'occurrence, il s'agit de leur date de naissance. On aboutit au tableau 12-4.

Tableau 12–4 Tableau descriptif complet des associations (liens) de notre exemple

Personne	Date de naissance	Rôle	*f*	Personne	Date de naissance	Rôle
Madame GR	19480908	Mère	Relation de parenté	Monsieur DH	19640316	Fils
Monsieur DH	19640316	Père	Relation de parenté	Monsieur JG	19861111	Fils
Madame GR	19480908	Grand-mère	Relation de parenté	Monsieur JG	19861111	Petit-fils

Cet exemple montre :

- qu'un lien peut être typé, « est le fils de » par exemple est un type de lien ;
- qu'une même ressource peut être alternativement source et cible de liens : son rôle varie ;
- que les informations doivent être factorisées pour éviter la duplication.

Nous allons voir comment XLink traduit ces concepts.

Revue des éléments de XLink servant à spécifier des liens

Modèle conceptuel de XLink

Avant de présenter le détail des possibilités offertes par les attributs définis par XLink, nous allons étudier son modèle conceptuel (figure 12-6).

La représentation UML offre une vue synthétique des combinaisons offertes par XLink. Elle a l'avantage d'être particulièrement simple à lire par rapport aux tableaux détaillés des sections qui suivent. Les deux types de liens XLink y sont représentés par les classes ExtendedLink et SimpleLink. Un lien étendu est composé de ressources locales (la classe `Ressource`), d'accès à des ressources distantes (la classe `Locator`) et de relations sémantiques entre les ressources (la classe `Arc`). XLink fait donc explicitement la différence entre les participants à un lien (les ressources locales ou distantes) et les caractéristiques sémantiques des relations entres les participants (les arcs). Il est ainsi possible d'associer plusieurs sémantiques à un même lien.

Le modèle conceptuel fait aussi observer qu'un lien simple regroupe sur un même élément les caractéristiques d'un lien étendu : il porte à la fois les caractéristiques d'accès à une ressource distante (`LocatorType`) et les caractéristiques sémantiques du lien (`ArcType`).

Figure 12–6
Modèle conceptuel
de XLink

Les paragraphes suivants décrivent le modèle XML physique de XLink. Nous disposons ainsi d'un autre exemple de transformation entre un modèle conceptuel et un modèle physique. Cette transformation est d'autant plus intéressante que XLink pousse à l'extrême les techniques d'utilisation d'attributs pour représenter la sémantique des éléments, tels que nous les avons présentés au chapitre 5, « Finaliser la sémantique du balisage ». Le modèle physique XLink se présente donc sous forme d'une liste d'attributs à placer sur les éléments XML. Les règles de placement de ces attributs sont directement issues du modèle conceptuel décrit à la figure 12-6.

Élément Xlink de type lien simple

Le plus basique des liens Xlink est celui de type simple. Ce lien a une source et une cible uniques. Les deux exemples qui suivent montrent la forme d'un lien simple.

Le premier exemple est l'expression la plus simple d'un lien Xlink. Les attributs `xmlns:xl`, `xl:type` et `xl:href` déclarent respectivement l'espace de noms de Xlink et le préfixe `xl` qui lui est associé, le type de lien et l'URL de la cible.

```
<logo xmlns:xl = "http://www.w3.org/1999/xlink" xl:type="simple"
xl:href="xmldev.gif" />
```

Dans le second exemple, les attributs `xl:show` et `xl:actuate` sont utilisés.

```
<logo xmlns:xl="http://www.w3.org/1999/xlink" xl:type="simple"
xl:href="xmldev.gif" xl:show="new" xl:actuate="onLoad"/>
```

Dans XLink, les éléments porteurs de liens simples n'ont pas d'enfant (au sens XLink), c'est-à-dire qu'ils n'ont aucun élément enfant spécifique dont la vocation serait de porter des informations complémentaires sur la nature du lien. En revanche, ces éléments peuvent avoir des enfants XML « normaux ».

Le tableau 12-5 liste les attributs Xlink autorisés sur les éléments Xlink de type simple.

Tableau 12–5 Attributs des éléments Xlink de type lien simple

Nom de l'attribut XLink	Type ou valeurs autorisées	Explications
type	"simple"	Cet attribut spécifie que le lien est de type simple.
href	CDATA	L'URI de la cible.
role	URI	Fournit l'URI d'une ressource qui indique le rôle du lien simple.
arcrole	URI	L'URI d'une ressource précisant la fonction du lien.
title	CDATA	Description du lien compréhensible par un être humain.
show	"new"	Le contenu de la cible s'affiche dans une nouvelle fenêtre.
	"replace"	Le contenu de la cible remplace celui de la fenêtre active.
	"embed"	Le contenu de la cible s'insère dans celui de la fenêtre active.
	"other"	Le type d'affichage n'est pas l'un des trois définis en standard mais reste toutefois spécifié dans le lien par un balisage.
	"none"	Le type d'affichage n'est pas l'un des trois définis en standard et n'est pas spécifié dans le lien par un quelconque balisage.
actuate	"onLoad"	Le lien est activé dès le chargement du document XML.
	"onRequest"	Le lien est activé par une action volontaire de l'utilisateur.
	"other"	Le type d'activation n'est pas l'un des deux cas standards mais reste toutefois spécifié dans le lien par un balisage.
	"none"	Le type d'activation n'est pas l'un des deux cas standards et n'est pas spécifié dans le lien par un quelconque balisage.

Le lien simple est comparable aux mécanismes des ID/IDREF de SGML et XML, et à l'élément `<a href="...">` de HTML.

Élément Xlink de type étendu (ou extended)

L'élément Xlink de type lien étendu permet de gérer des liens plus sophistiqués que le lien simple. Il s'agit notamment de ceux dont la cible est définie de manière logique, mais il peut s'agir plus simplement d'un lien pointant sur plusieurs cibles.

Un lien étendu permet d'associer autant de ressources que nécessaire.

Quand les ressources associées par le lien sont locales, on les définit à l'aide d'éléments de type « ressource », tandis que dans le cas où elles sont distantes, on les définit à l'aide d'éléments de type « localiseur ».

D'une manière générale, l'élément de type étendu s'accompagne des éléments enfants de types « localiseur », « arc », « ressource » et « titre ».

Nous donnons ci-après un exemple de lien étendu.

```
<extendedlink xmlns:xl="http://www.w3.org/1999/xlink"
xl:type="extended">
<ressource-locale xl:type="resource">
       n'importe quel contenu ou vide
</ressource-locale>
 <ressource-distante xl:type="locator" xl:href="http://www.dev.com/
index.html">
Attribut href obligatoire sur élément de type locator et l'élément
parent doit être de type extended
</ressource-distante>
</extendedlink>
```

Le tableau 12-6 liste les attributs autorisés sur les éléments Xlink de type étendu. Dans notre exemple, nous pourrions par exemple avoir :

```
<extendedlink xmlns:xl="http://www.w3.org/1999/xlink"
xl:type="extended" role=" http://www.dev.com/indexDescription.html"
title="Lien entre un index et un mot">
```

Dans cet exemple, on est censé trouver à l'URI spécifié par l'attribut `role` une description compréhensible par l'application du rôle du lien.

Tableau 12–6 Attributs des éléments Xlink de type étendu

Nom de l'attribut XLink	Type ou valeurs autorisées	Explications
type	"extended"	Cet attribut spécifie que le lien est de type étendu.
role	URI	Fournit l'URI d'une ressource renseignant sur le rôle du lien étendu.
title	CDATA	Description compréhensible de l'élément de type lien étendu.

Élément Xlink de type ressource

Les éléments de type ressource servent à définir des ressources locales, a priori destinées à participer à un lien étendu.

Les ressources locales sont principalement identifiées grâce à un label, ou identifiant unique.

Les ressources locales n'ont pas d'élément Xlink enfant particulier.

Le tableau 12-7 liste les attributs autorisés sur les éléments Xlink de type ressource.

Tableau 12–7 Attributs des éléments Xlink de type ressource

Nom de l'attribut XLink	Type ou valeurs autorisées	Explications
type	"resource"	Cet attribut spécifie que le lien est de type « resource ».
role	URI	URI où l'on pourra trouver le rôle de la ressource locale.
title	CDATA	Description compréhensible de la ressource locale.
label	NCName	Identifiant unique de la ressource.

Voici un exemple de ressource locale, qui appelle un graphique au moyen d'un lien simple :

```
<figure xl:type="resource" xl:label="fig001"
        xmlns:xl="http://www.w3.org/1999/xlink">
<légende>LH Multifunction Head Down Display</légende>
<graphic xl:type="simple"
         xl:href="ICN-1B-B-311501-B-K0999-00352-A-01-1.CGM"
         xl:actuate="onLoad xl:show="embed"/>
</figure>
```

Élément Xlink de type localiseur (ou locator)

Les éléments de type localiseur servent à spécifier les adresses des ressources distantes qui participent à un lien.

Le tableau 12-8 liste les attributs autorisés des éléments Xlink de type localiseur.

Tableau 12–8 Attributs des éléments Xlink de type localiseur

Nom de l'attribut XLink	Type ou valeurs autorisées	Explications
Type	"Locator"	Cet attribut spécifie que le lien est de type localiseur.
Role	URI	URI où l'on pourra trouver le rôle de la ressource distante.
Title	CDATA	Description compréhensible du localiseur.

Tableau 12–8 Attributs des éléments Xlink de type localiseur

Nom de l'attribut XLink	Type ou valeurs autorisées	Explications
Label	NCName	Identifiant du localiseur.
Href	URI	Attribut obligatoire qui fournit l'URI d'une ressource dite distante.

Cet élément accepte un seul enfant Xlink : celui de type `title`.

On pourrait souhaiter transformer en élément de type localiseur l'exemple de type ressource précédent, ce qui donnerait :

```
<figure xl:type="locator" xl:label="fig001" xmlns:xl="http://
www.w3.org/1999/xlink"
        xl:href="ICN-1B-B-311501-B-K0999-00352-A-01-1.CGM">
<légende xl:type="title">LH Multifunction Head Down Display</légende>
</figure>
```

Dans cet exemple, nous avons transformé la combinaison d'une ressource et d'un lien simple en un seul lien élément de type localiseur. On a simplifié le balisage, mais au détriment de la possibilité de spécifier directement le comportement que l'application doit avoir au moment de l'activation du lien : on a en outre la possibilité de spécifier les attributs `xl:actuate` et `xl:show`.

Élément Xlink de type arc

L'élément de type arc sert à définir une relation entre deux ressources, locales ou distantes, et donc de type ressource ou localiseur. La relation est définie par trois éléments : un type, une source et une cible. Dans une relation, la source et la cible jouent chacune un rôle par rapport à la nature de la relation (c'est ce qui va nous permettre d'exprimer les différents cas de figure des exemples présentés au tableau 12-4).

Voici un premier exemple de mise en œuvre d'un élément de type arc, dans lequel on voit bien les notions de type de relation et de rôle des source et cible :

```
<relation xmlns:xl=http://www.w3.org/1999/xlink
     xl:type="arc"
     xl:from="zeus"
     xl:to="heracles"
     xl:arcrole="fils"/>
<god xl:href="greece/gods/gods.xml#god[@name='zeus']"
     xl:label="zeus"
     xl:role="dieu grec"
     xl:title="Zeus"
     xl:type="locator"/>
```

```
<god xl:href="greece/gods/gods.xml#god[@name='heracles']"
     xl:label="heracles"
     xl:role="demi-dieu grec"
     xl:title="Heracles"
     xl:type="locator"/>
```

Le tableau 12-9 liste les attributs autorisés des éléments Xlink de type arc.

Tableau 12–9 Attributs des éléments Xlink de type arc

Nom de l'attribut XLink	Type ou valeurs autorisées	Explications
type	"arc"	Cet attribut spécifie que le lien est de type arc.
arcrole	URI	Le rôle de la ressource ciblée par l'attribut `to` dans l'arc
title	CDATA	
show	"new"	Le contenu de la cible s'affiche dans une nouvelle fenêtre.
	"replace"	Le contenu de la cible remplace celui de la fenêtre active.
	"embed"	Le contenu de la cible s'insère dans celui de la fenêtre active.
	"other"	Le type d'affichage n'est pas l'un des trois définis en standard, mais reste toutefois spécifié dans le lien par un balisage.
	"none"	Le type d'affichage n'est pas l'un des trois définis en standard et n'est pas spécifié dans le lien par un quelconque balisage.
actuate	"onLoad"	Le lien est activé dès le chargement du document XML.
	"onRequest"	Le lien est activé par une action volontaire de l'utilisateur.
	"other"	Le type d'activation n'est pas l'un des deux cas standards mais reste toutefois spécifié dans le lien par un balisage.
	"none"	Le type d'activation n'est pas l'un des deux cas standards et n'est pas spécifié dans le lien par un quelconque balisage.
from	NCName	
to	NCName	Le label d'une ressource locale ou élément de type localiseur.

Cet élément accepte un seul enfant Xlink : celui de type `title`.

Élément Xlink de type titre (ou title)

L'élément de type titre permet d'exprimer des labels lisibles et compréhensibles par l'œil humain, nous permettant ainsi de reconnaître lorsqu'ils sont affichés, le type du lien ou de la ressource, consulté ou activé.

Le tableau 12-10 liste les attributs autorisés des éléments Xlink de type titre.

Cet élément n'admet pas d'autre élément Xlink enfant.

Tableau 12–10 Attributs des éléments Xlink de type titre

Nom de l'attribut XLink	Type ou valeurs autorisées	Explications
type	"title"	Cet attribut spécifie que le lien est de type titre.

Exemple de mise en œuvre de XLink

Lien de type simple

Dans la version XML Schema de la norme S1000D, déjà présentée plusieurs fois dans les chapitres précédents, les liens à des ressources externes sont de type XLink.

Les ressources externes sont des fichiers graphiques, des modules XML et des publications.

Dans l'exemple ci-après, un lien Xlink est établi avec un module XML. Il s'agit d'un lien simple utilisant l'URI du module cible (*via* l'attribut `xl:href`) également accompagné de la définition logique de ce module (*via* l'élément `avee`) :

```
<refdm xmlns:xl="http://www.w3.org/1999/xlink"
  xl:type="simple"
  xl:actuate="onRequest"
  xl:show="replace"
  xl:title="Interactive Electronic Technical Publications -
           ➥XML-oriented (IETP-X)"
  xl:href="http://www.aecma.org/Publications/Spec1000D/
           ➥DMC-AE-A-00-40-05-50A-000A-A.XML">
<avee>
<modelic>AE</modelic>
<sdc>A</sdc>
<chapnum>00</chapnum>
<section>4</section>
<subsect>0</subsect>
<subject>05</subject>
<discode>50</discode>
<discodev>A</discodev>
<incode>000</incode>
<incodev>A</incodev>
<itemloc>A</itemloc>
</avee>
</refdm>
```

Dans ce premier exemple, on remarquera que le lien simple porte sur l'ensemble du bloc de données avee et non sur un seul de ces éléments. D'où l'intérêt de décrire un lien simple sur un élément qui peut lui-même contenir d'autres éléments.

Le deuxième exemple montre un lien vers une illustration où l'on retrouve un mécanisme similaire de double identification de la cible : physique au travers de l'attribut xl:href et logique au travers de l'attribut boardno :

```
<figure id="fig001">
<title>LH Multifunction Head Down Display</title>
<graphic xmlns:xl="http://www.w3.org/1999/xlink" xl:type="simple"
         xl:href="ICN-1B-B-311501-B-K0999-00352-A-01-1.CGM"
         xl:actuate="onLoad"
         xl:show="embed"
         boardno="ICN-1B-B-311501-B-K0999-00352-A-01-1"/>
</figure>
```

Dans cet exemple, la seule utilité du double lien est qu'il fournit simultanément les adresses physique et logique d'une illustration. Cette redondance peut sembler inutile mais, dans la pratique, il est souvent utile de dissocier les identifiants logiques d'illustrations des adresses physiques des fichiers les contenant, un peu à la manière de ce qui a été présenté en début de chapitre.

Le principe est le même pour cibler un élément particulier à l'intérieur d'un module de données. Dans l'exemple suivant, on cible l'élément dont l'identifiant est CSN-53251001-001-00A :

```
<csnref xmlns:xl="http://www.w3.org/1999/xlink"
        xl:type="simple"
        xl:href="DMC-A1-A-53-25-10-010-941A-A_003.XML#
                 ➥CSN-53251001-001-00A"
        xl:actuate="onRequest"
        xl:show="new"
        refcsn="53251001 001"
        refisn="00A"/>
```

Lien de type complexe

Le cas pratique présenté au tableau 12-4 sera exprimé par des liens XLink de type arc. Dans l'exemple présenté ci-après, les éléments personne, objet et relation ne font pas partie du vocabulaire de XLink. Quant au préfixe xl, c'est celui habituellement associé à l'espace de noms de Xlink.

```
<personne id="gr" naissance="19480908">Mme Gertrude R</personne>
<personne id="dh" naissance="19640316">Mr Denis H</personne>
<personne id="jg" naissance="19861111">Mr Jérome G</personne>

<lienparent xl:type="simple" role="relation de filiation">
  <part xl:type="locator" xl:label="gertrude" role="proche"
        xl:href="#personne[@id='gr']"/>
  <part xl:type="locator" xl:label="denis" role="proche"
        xl:href="#personne[@id='dh']"/>
  <relation xl:type="arc" xl:from="gertrude" xl:to="denis"
            xl:arcrole="fils"/>
  <relation xl:type="arc" xl:from="denis" xl:to="gertrude"
            xl:arcrole="mère"/>
</lienparent>

<lienparent xl:type="simple" role="relation de filiation">
  <part xl:type="locator" xl:label="denis" role="proche"
        xl:href="#personne[@id='dh']"/>
  <part xl:type="locator" xl:label="jerome" role="proche"
        xl:href="#personne[@id='jg']"/>
  <relation xl:type="arc" xl:from="denis" xl:to="jerome"
            xl:arcrole="fils"/>
  <relation xl:type="arc" xl:from="jerome" xl:to="denis"
            xl:arcrole="père"/>
</lienparent>

<lienparent xl:type="simple" role="relation grandparentale">
  <part xl:type="locator" xl:label="gertrude" role="proche"
        xl:href="#personne[@id='gr']"/>
  <part xl:type="locator" xl:label="jerome" role="proche"
        xl:href="#personne[@id='jg']"/>
  <relation xl:type="arc" xl:from="gertrude" xl:to="jerome"
            xl:arcrole="petit-fils"/>
  <relation xl:type="arc" xl:from="jerome" xl:to="gertrude"
            xl:arcrole="grand-mère"/>
</lienparent>
```

On pourrait tout aussi bien créer un seul lien complexe comme indiqué ci-après.

```
<lienparent xl:type="simple" role="relation de parenté">
  <part xl:type="locator" xl:label="gertrude" role="proche"
        xl:href="#personne[@id='gr']"/>
  <part xl:type="locator" xl:label="denis" role="proche"
        xl:href="#personne[@id='dh']"/>
```

```
  <part xl:type="locator" xl:label="jerome" role="proche"
        xl:href="#personne[@id='jg']"/>
  <relation xl:type="arc" xl:from="gertrude" xl:to="denis"
            xl:arcrole="fils"/>
  <relation xl:type="arc" xl:from="denis" xl:to="gertrude"
            xl:arcrole="mère"/>
  <relation xl:type="arc" xl:from="denis" xl:to="jerome"
            xl:arcrole="fils"/>
  <relation xl:type="arc" xl:from="jerome" xl:to="denis"
            xl:arcrole="père"/>
  <relation xl:type="arc" xl:from="gertrude" xl:to="jerome"
            xl:arcrole="petit-fils"/>
  <relation xl:type="arc" xl:from="jerome" xl:to="gertrude"
            xl:arcrole="grand-mère"/>
</lienparent>
```

On peut observer que nous avons ici un document XML traduisant parfaitement les associations souhaitées, mais que seul un programme peut pratiquement exploiter. En l'occurrence, il faut remarquer que ni ce document XML ni son schéma associé ne peuvent exprimer la sémantique préconisée par leur auteur. Seule la représentation UML du modèle conceptuel peut transmettre cette information.

L'approche de l'ISO : le modèle Topics Map

Dans cette section, nous présentons la spécification XTM (XML Topics Map), qui est une proposition de vocabulaire XML pour les Topics Maps, ou cartes de topiques, qui fait l'objet de la norme ISO 13250 de 2000. Les cartes de topiques sont des ensembles de ressources hautement interconnectées entre elles par des faisceaux de relations. Il s'agit d'un modèle de liens qui adresse un niveau supérieur à Xlink, et, à ce titre, englobe cette recommandation du W3C.

> **SPÉCIFICATION XTM XML Topics Map**
>
> Produit par TopicMaps.Org, le texte originel se trouve à l'URL http://www.topicmaps.org/xtm/1.0/xtm1-20010806.html et sa version française est hébergée par XMLfr.org (http://www.xmlfr.org).

XTM propose un modèle qui est apprécié de ceux qui travaillent dans les mondes de l'édition, de la documentation, des bibliothèques, de l'industrie et de la gestion des connaissances. Il apparaît comme plus concret, pragmatique et facile à mettre en œuvre que RDF ; il est vrai qu'il bénéficie d'une expérience du terrain supérieure de

plusieurs années à celle qui accompagne RDF, et que ses concepteurs ne viennent pas du monde de l'intelligence artificielle mais de celui des documents où naquirent aussi la plupart des modèles opérationnels ayant trait à la structuration de l'information. En voici un bref historique :

- 1985 : Ch. Goldfarb lance une série de travaux sur la description structurée et synchronisée de la musique, travaux connus sous le nom de Standard Music Description Language, acronyme SMDL. De ces travaux émergèrent les concepts et formulations concrètes des liens hypermédias.
- 1991 : le groupe de travail Davenport commence les travaux sur le standard Sofabed (Standard Open Formal Architecture for Browsable Electronic Documents).
- 1992 : naissance de la norme ISO HyTime (Hypermedia Time based Structuring Language – ISO 10744). Elle établit un modèle de liens hypertextuels internes et externes aux documents dont les extrémités peuvent être de nature et de granularité diverses. HyTime permet également d'exprimer des contraintes de synchronisation temporelle entre des événements de type vidéo, son, séquences photos...
- 1994 : HyTime, trop complexe à mettre en œuvre tel quel, donne lieu à l'écriture de conventions d'application simplifiées baptisées CApH (Conventions for the Applications of HyTime). Conduits par Steve Newcomb et Michel Biezunski, ces travaux sont à la base de l'invention de Topics Map.
- Octobre 1996, formation d'un groupe de travail sur Topics Map au sein de l'ISO, qui aboutit au standard ISO/IEC 13250 en janvier 2000. Le standard repose sur SGML et utilise les formes architecturales de HyTime. La syntaxe HyTime concrète utilisée dans Topics Map est baptisée HyTM, pour HyTime Topics Map.
- Décembre 2001 : la version XML de Topics Map devient une norme ISO. Cette version XML ne se contente pas de formuler un modèle conforme à XML 1.0, mais encore spécifie l'adressage des éléments externes au moyen d'URI. D'autres différences, notamment de structure des modèles de contenu, font que les instances HyTM et les documents XTM sont incompatibles. Un convertisseur est nécessaire pour passer de l'un à l'autre.

Un des points les plus intéressants de la série des standards HyTime et Topics Map est l'utilisation de *formes architecturales*, concept qui consiste à proposer des vocabulaires XML génériques destinés à être embarqués dans d'autres. La sémantique de ces vocabulaires l'emporte alors sur la forme XML physique : XTM, par exemple, est l'expression physique des formes architecturales définies par Topics Map.

Le concept de cartes de topiques, traduction française de Topics Map, repose sur une séparation nette opérée entre le fait de relier des éléments entre eux et celui de les localiser. Nous avons déjà expliqué ce concept dans ce chapitre. Cela a pour effet principal de rendre la pose de liens totalement indépendante des supports physiques, permettant ainsi d'obtenir des milliers de vues différentes d'une même base de connaissances.

Brève description

Les cartes de topiques sont comme les cartes routières : les villes y sont remplacées par des topiques et les routes par des relations. Les topiques ont des caractéristiques qui permettent de les identifier, et les relations sont typées. La comparaison avec les cartes routières est, à ce stade, très aisée. La mise en œuvre est un peu plus complexe car les topiques et les relations sont, dans la vie réelle, polymorphes : leur sens peut varier en fonction d'un contexte. Ils sont abstraits et se situent dans un espace virtuel à n dimensions. La comparaison avec les cartes routières ne peut, sur ce point, perdurer.

VOCABULAIRE Topiques

En français, un topique est un sujet mis dans un contexte. Par exemple, il est plus élégant, en introduction d'une intervention publique, de présenter ses *topiques* plutôt que ses *sujets*. On aurait pu traduire *Topics Map* par *carte de sujets*, mais il est plus juste d'utiliser l'expression *carte de topiques*, justement parce que, dans le concept développé par la norme Topics Map, le sens exact des sujets dépend de leur contexte d'utilisation. Il s'agit donc bien de topiques.

VOCABULAIRE Topique, sujet et réification

Comme nous l'avons dit, un topique est un sujet mis dans un contexte. Il est donc possible d'isoler d'un côté des sujets et de l'autre des topiques. Quand on met un sujet dans un certain contexte, il devient un topique : on dit que le sujet est réifié. La réification est un processus important du concept de carte de topiques. Nous ne nous étendrons toutefois pas dessus dans ce chapitre. Pour plus de précision, référez-vous à la version française ou anglaise de la spécification XTM que vous trouverez respectivement aux adresses XMLfr.org et http://www.topicmaps.org/xtm/1.0/xtm1-20010806.html.

Une des originalités de Topics Map est que tout y est topique, y compris les relations. Nous aurons l'occasion de revenir sur ce point.

Dans l'exemple suivant, trois topiques sont définis (repères ⓿, ❶ et ❷) :

```
<?xml version="1.0" ?>
<topicMap xmlns="http://www.topicmaps.org/xtm/1.0/"
          xmlns:xl="http://www.w3.org/1999/xlink">
 <topic id="famille"> ← ⓿
  <baseName>
   <baseNameString>Famille</baseNameString>
  </baseName>
 </topic>
 <topic id="personne"> ← ❶
  <baseName>
   <baseNameString>Personne</baseNameString>
  </baseName>
```

```
  </topic>
  <topic id="jean-chauveau"> ← ❷
   <instanceOf>
    <topicRef xl:href="#personne"/> ← ❸
   </instanceOf>
   <baseName>
    <baseNameString>Jean chauveau</baseNameString>
   </baseName>
  </topic>
</topicMap>
```

Le topique ❷ est défini comme étant une instance du ❶ via l'élément `instanceOf`. Cette relation indique que le topique connu grâce à l'identifiant `Jean Chauveau` fait également partie de la classe de ceux connus *via* l'identifiant `Personne`. On voit que le mécanisme de référencement des topiques (repère ❸) s'appuie sur un lien Xlink de type simple.

Dans notre exemple, le topique `Personne` ne fait pas référence à un autre sujet (le chaînage des topiques devant bien s'arrêter un jour !) ; son seul topique parent est la classe des topiques génériques. Ce topique se trouve ainsi être à la tête de tous les topiques obtenus par instanciation de ce topique (via l'élément `instanceOf`). C'est ainsi que se définissent des classes de topiques.

Topics Map permet d'associer des topiques, dans ce cas appelés ancres, en suivant des relations logiques. Les topiques ainsi associés ne sont pas obligatoirement interchangeables (par exemple, si `a` est le fils de `b`, l'inverse n'est évidemment pas vrai). Pour chaque association, les topiques jouent un rôle particulier.

Topics Maps tient compte de ce que les choses peuvent être exprimées de différentes manières, par exemple : *le 16e siècle*, *le XVIe siècle* et *la Renaissance* sont trois titres de topiques représentant la même époque historique.

Les concepts développés par Topics Map peuvent être utilisés pour représenter des index, des glossaires, des références croisées. Ainsi, un glossaire est constitué d'ancres associées deux par deux : un mot ou une expression (une première ancre) est associé à une définition (une seconde ancre) par une relation de type *éléments de glossaire*.

Relation entre topiques et ressources physiques

Il est inutile de définir des ressources logiques et des cartes de topiques si, au bout du compte, on ne tombe pas sur un objet bel et bien concret ; quand on cherche un texte de loi, on souhaite, au terme de la recherche, aboutir soit au texte de loi lui-même, soit à un texte explicatif si seule la version papier du texte existe (par exemple). Topics Map permet de faire la distinction entre :

- une ressource informatique accessible qui est alors l'objet terminal du topique ;

- une ressource physique inaccessible de manière informatique qui oblige à passer par une ressource dérivée.

Topics Map décrit ainsi des objets du monde réel, par exemple un sapin. Il est évidemment impossible de mettre physiquement l'arbre dans l'ordinateur, il faudra utiliser des ressources palliatives telles qu'une page Web, une photo ou, plus généralement, les objets dans lesquels cet arbre apparaît ou est cité. Cela peut aussi bien être les coordonnées géodésiques du sapin en question !

En revanche, si le sujet décrit une ressource telle qu'un manuel de réparation, alors il est à peu près certain que le document électronique correspondant sera accessible informatiquement, et la ressource associée au sujet sera ce document informatique. Il s'agit d'une ressource terminale.

Nous donnons un exemple ci-après de ces deux cas de figure.

Tout d'abord, voyons celui de la ressource terminale. L'élément `ResourceRef` est utilisé pour en fournir l'URI :

```
<subjectIdentity>
 <ResourceRef xl:href="http://www.chauveau.com/genealogie/jean.html"/>
</subjectIdentity>
```

Il ne peut y avoir qu'une seule ressource terminale par topique.

Ensuite, examinons celui de la ressource dérivée ; l'élément `subjectIndicatorRef` est utilisé pour désigner la ressource :

```
<subjectIdentity>
 <subjectIndicatorRef
        ➥xl:href="http://www.familles.com/chauveau/index.html"/>
 <subjectIndicatorRef
        ➥xl:href="http://www.opera.com/rouen/chanteurs/index.html"/>
</subjectIdentity>
```

Dans ce cas, il peut y avoir plusieurs ressources dérivées.

Les deux types de ressources peuvent être combinés autant que nécessaire, et notre exemple pourrait être :

```
<?xml version="1.0" ?>
<topicMap xmlns="http://www.topicmaps.org/xtm/1.0/"
          xmlns:xl="http://www.w3.org/1999/xlink">
<topic id="personne">
  <baseName>
   <baseNameString>Personne</baseNameString>
```

```
  </baseName>
</topic>
<topic id="jean-chauveau">
  <instanceOf>
   <topicRef xl:href="#personne" />
  </instanceOf>
  <subjectIdentity>
   <subjectIndicatorRef
        ➥xl:href="http://www.familles.com/chauveau/index.html"/>
  <subjectIndicatorRef
        ➥xl:href="http://www.opera.com/rouen/chanteurs/index.html"/>
  <resourceRef xl:href="http://www.chauveau.com/genealogie/jean.html"/>
  </subjectIdentity>
  <baseName>
   <baseNameString>Jean Chauveau</baseNameString>
  </baseName>
</topic>
</topicMap>
```

Les associations de topiques

Les topiques pouvant eux-mêmes être des liens, le fait d'associer certains topiques entre eux consiste à établir des liens de liens. Ce faisant, on s'aperçoit qu'il est possible de définir son propre faisceau de liens indépendamment (ou presque) des topiques originels. Plus exactement, il est possible de définir ses propres relations sans être aucunement contraint par la définition initiale que le topique exprime.

Dans l'exemple suivant, les topiques `jean-chauveau` et `operaRouen` sont associés par la relation `est-membre`. Ils y jouent respectivement les rôles définis par les topiques `chanteur` et `chœurs`.

```
<association id="est-membre">
 <instanceOf>
  <topicRef xl:href="#est-chanteur"/>
 </instanceOf>
 <member>
  <roleSpec>
   <topicRef xl:href="#chanteur"/>
  </roleSpec>
  <topicRef xl:href="#jean-chauveau"/>
  </member>
 <member>
  <roleSpec>
   <topicRef xl:href="#choeurs"/>
```

```
    </roleSpec>
    <topicRef xl:href="#operaRouen"/>
  </member>
</association>
```

Et, pour que notre association fonctionne, il faut déclarer les topiques qui s'y trouvent utilisés :

```
<?xml version="1.0" ?>
<topicMap xmlns="http://www.topicmaps.org/xtm/1.0/" xmlns:xl="http://
www.w3.org/1999/xlink">
<topic id="personne">
  <baseName>
   <baseNameString>Personne</baseNameString>
  </baseName>
</topic>
<topic id="jean-chauveau"> ← ❶
  <instanceOf>
   <topicRef xl:href="#personne" />
  </instanceOf>
  <subjectIdentity>
   <subjectIndicatorRef xl:href="http://www.familles.com/chauveau/
index.html"/>
   <subjectIndicatorRef xl:href="http://www.opera.com/rouen/chanteurs/
index.html"/>
   <ResourceRef xl:href="http://www.chauveau.com/genealogie/jean.html"/
>
  </subjectIdentity>
  <baseName>
   <baseNameString>Jean Chauveau</baseNameString>
  </baseName>
</topic>
<topic id="chanteur">
 <baseName>
  <baseNameString>Chanteur</baseNameString>
 </baseName>
</topic>
<topic id="choeurs"> ← ❶
 <baseName>
  <baseNameString>Choeurs d'opéra</baseNameString>
 </baseName>
</topic>
<topic id="opera">
 <baseName>
  <baseNameString>Opéra</baseNameString>
```

```
 </baseName>
</topic>
<topic id="operaRouen">
 <instanceOf>
  <topicRef xl:href="#opera" />
 </instanceOf>
 <baseName>
  <baseNameString>Opéra de rouen</baseNameString>
  </baseName>
</topic>
<topic id="est-choriste"> ← ❷
  <baseName>
   <baseNameString>Est membre de</baseNameString>
  </baseName>
  <baseName>
   <scope>
    <topicRef xl:href="#choeurs"/> ← ❹
   </scope>
   <baseNameString>Un choeur</baseNameString>
  </baseName>
  <baseName>
  <scope>
    <topicRef xl:href="#chanteur"/> ← ❺
   </scope>
  <baseNameString>Un chanteur</baseNameString>
  </baseName>
</topic>
<association id="jean-chauveau-est-choriste-de"> ← ❸
  <instanceOf>
   <topicRef xl:href="#est-choriste"/>
  </instanceOf>
  <member>
   <roleSpec>
    <topicRef xl:href="#chanteur"/>
   </roleSpec>
   <topicRef xl:href="#jean-chauveau"/>
  </member>
  <member>
   <roleSpec>
    <topicRef xl:href="#choeurs"/>
   </roleSpec>
   <topicRef xl:href="#operaRouen"/>
  </member>
</association>
</topicMap>
```

Ainsi, on associe (❸) deux topiques (⓿et ❶) selon une relation (❷), elle-même définie par des topiques (❹ et ❺).

Nous parvenons à un stade où une représentation graphique s'impose. La figure 12-7 montre le schéma de relations que notre Topics Map exprime sous forme XML.

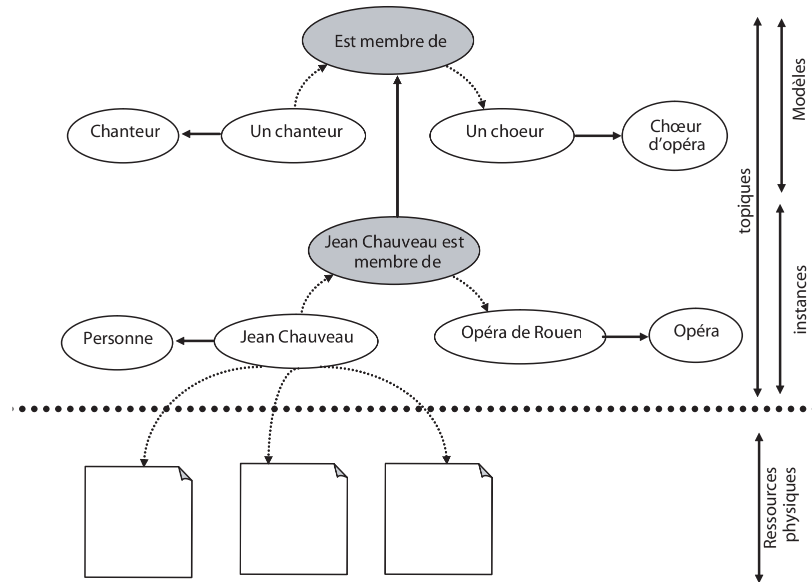


Figure 12–7
Représentation graphique de la carte de topiques de notre exemple

En clair, voici ce que signifie, dans cet exemple, notre carte de topiques (Topics Map) :

- Le sujet `Jean Chauveau` est une instance du sujet `Personne`.
- Le sujet `Opéra de Rouen` est une instance du sujet `Opéra`.
- Le sujet `Jean Chauveau` permet d'accéder à trois ressources physiques.
- Le sujet `Est membre de` associe `Un chanteur` et `Un chœur`, qui doivent être respectivement des topiques des classes `Chanteur` et `Chœur d'opéra`.
- Le sujet `Jean Chauveau` est associé au sujet `Opéra de Rouen` au moyen du sujet Jean Chauveau `est membre de`. Cette association est une instance du sujet `Est membre de`. Sa définition nous permet de savoir que, dans ce cas présent, `Jean Chauveau` joue le rôle d'`Un chanteur`, et `Opéra de Rouen` celui d'`Un chœur`.

Jusqu'à présent, nous avons pu classer nos topiques et créer entre eux des relations. Les topiques et les associations peuvent être instanciés. Nous verrons à la section suivante le troisième et dernier concept intervenant dans une carte de topiques : celui d'occurrence de topiques.

VOCABULAIRE Instancier

Une définition communément admise est : « *verbe transitif, créer une* instance *d'une* classe, *c'est-à-dire un objet* » (référence : *Le Jargon français* — dictionnaire d'informatique francophone — http://www.linux-france.org/prj/jargonf/I/instancier.html).
La définition officielle à retenir est celle établie en 2002 dans le grand dictionnaire terminologique de l'Office québécois de la langue française : « En programmation orientée objet, créer à partir d'une classe une occurrence de cette classe, héritant par défaut des attributs de sa classe, et qui peut être dotée d'attributs spécifiques » (http://www.granddictionnaire.com).

Les occurrences de topiques

La notion d'occurrence est très proche de celle déjà mentionnée de référencement de ressources physiques *via* les éléments `subjectIndicatorRef` et `ressourceRef`.

La différence tient à ce que le concept d'occurrence indique en plus la nature du support physique référencé, précisément en référençant un topique...

Par exemple, pour indiquer que `Jean Chauveau` apparaît sur une photo disponible en ligne dont le nom de fichier est `JC1921.gif`, on écrira :

```
<topic id="jean-chauveau">
 <instanceOf>
  <topicRef xl:href="#personne" />
 </instanceOf>
 <subjectIdentity>
  <subjectIndicatorRef
        ➥xl:href="http://www.familles.com/chauveau/index.html"/>
  <subjectIndicatorRef
        ➥xl:href="http://www.opera.com/rouen/chanteurs/index.html"/>
  <ResourceRef xl:href="http://www.chauveau.com/genealogie/jean.html"/>
 </subjectIdentity>
 <baseName>
  <baseNameString>Jean Chauveau</baseNameString>
 </baseName>
<occurrence>
  <instanceOf>
   <topicRef xl:href="#photo"/>
  </instanceOf>
  <ResourceRef xl:href="JC1921.gif" />
 </occurrence>
</topic>
```

La différence sémantique entre les éléments `subjectIdentity` et `occurrence` apparaît clairement : le premier renvoie à des ressources qui décrivent le sujet, tandis que le second indique les endroits physiques où le sujet apparaît.

Carte de topiques déduites du balisage

Une carte de topiques peut être produite à partir d'ensembles de données structurés comme en présentent les documents XML et pour lesquels les relations entre les topiques sont formellement ou implicitement décrites.

Il est intéressant d'avoir à l'esprit qu'une carte de topiques existe implicitement dans tout document XML contenant un balisage sémantique de l'information. Pour montrer cela, nous allons utiliser l'exemple suivant :

```
<auteur>
 <prénom>Jean</prénom>
 <nom>Noël</nom>
 <fonction>Directeur</fonction>
 <adresse>
  <société>CAEDIS</société>
  <ville>Bois-Guillaume</ville>
  <pays>France</pays>
  <tél>+33 235604289</tél>
  <fax>+33 235604290</fax>
  <email>jnoel@caedis.com</email>
  <siteweb>www.caedis.com</siteweb>
 </adresse>
</auteur>
```

Ce balisage identifie :

- Des classes d'information telles que `auteur`, `société`, `ville`, `pays`... À chaque élément peut correspondre un topique.
- Des associations : `auteur/société`, `société/ville`, `ville/pays`, `auteur/téléphone`, etc.
- Des occurrences : les valeurs contenues dans le balisage. Chaque donnée peut être l'occurrence du topique correspondant à l'élément encadrant la donnée.

La carte de topiques ainsi déduite de ce fragment XML devra toutefois être terminée manuellement au cas où les topiques devraient être hiérarchisés.

Lors de la récupération automatique des données à partir d'un document XML, l'une des tâches consiste à lisser le résultat obtenu, notamment les occurrences. Par exemple, deux numéros de téléphone identiques peuvent ne pas avoir été écrits tout à fait de la même manière, *idem* pour les noms de ville (par exemple, Bois-Guillaume *versus* Bois Guillaume sans trait d'union). Il se peut qu'un seul document XML (et c'est même certain) ne représente pas à lui tout seul tous les cas que l'on voudrait modéliser avec une carte de topiques.

Ce petit exemple montre déjà l'étendue des possibilités et les ouvertures offertes par l'approche Topics Map :

- Tous les éléments ne sont pas candidats à devenir des topiques ou des associations. Par exemple, il serait idiot de produire des associations `tél/ville` ou `mail/tél`. La classification des topiques finalement définie représentera une vision différente, souvent plus large, de celle fournie initialement par le document XML ayant servi à la construire.
- Des documents XML ayant des vocabulaires différents pourront facilement être mis en correspondance avec une Topics Map existante. Il suffira pour cela d'écrire les règles de correspondance des topiques.
- Inversement, des catalogues d'informations non balisées pourront être confrontés à notre Topics Map. Par exemple, si le nom `Jean Noël` est repéré dans un fichier informatique, on pourra espérer trouver rapidement ce que la carte de topiques contient, de manière structurée, à son sujet (sans jeu de mot !).

Questions relatives à la gestion des liens

Dans cette section, nous allons étudier les conséquences des documents massivement hyperliés en termes de gestion des contenus qui touchent les mécanismes :

- de gestion des révisions,
- de gestion des liens,
- de gestion des documents.

La gestion des révisions pose un problème tout à fait particulier dans le cas des documents hyperliés et nous allons l'expliquer. Notamment, nous allons montrer comment l'augmentation du nombre de liens peut emprisonner peu à peu un système de gestion de documents.

Le problème de la gestion des liens apparaît dès que les documents doivent être gérés en configuration.

Enfin, la gestion du document devient problématique dès qu'il n'est plus qu'un réceptacle de données stockées à l'extérieur. Nous allons expliquer ce qui se passe dans ce cas.

Liens et gestion des révisions

Dans la plupart des environnements professionnels, les impératifs de traçabilité de l'information et de sécurité (par l'identification de l'information) imposent des contraintes fortes sur les documents électroniques.

Dans le monde du Web et des pages HTML, il existe encore peu de contraintes (et de possibilités) d'affichage des révisions, mais dans les versions papier et électroniques des documents sensibles, ces contraintes sont particulièrement prégnantes et difficiles à respecter. Une seule donnée modifiée dans un système de gestion et c'est potentiellement une cascade de mises à jour qu'il faut effectuer : republication des documents avec mise en valeur de la donnée modifiée, repérage exact de la page modifiée, production d'une *liste effective des pages*, déclenchement d'une demande de mise à jour de tel ou tel autre document, etc.

La mise à jour d'un seul document peut avoir des répercussions sur les processus et entraîner la modification d'un certain nombre d'autres documents, en particulier ceux qui lui sont liés. C'est ainsi qu'il existe, comme nous allons le voir, un risque non négligeable de paralysie d'un référentiel documentaire.

Le risque : la paralysie du système

Tout le problème part de ce que les liens sont généralement définis au moyen d'identifiants uniques : un élément XML est pourvu d'un identifiant que l'on référence ailleurs, complété du chemin d'accès en dur au fichier qui le contient. Le problème se pose de la même façon, que les identifiants soient des ID/IDREF ou des URL.

Or, la contrainte de traçabilité crée la situation cornélienne suivante : pour conserver l'ancienne version d'une donnée XML, il n'y a que deux possibilités : la duplication de la totalité du document XML contenant la donnée modifiée ou la duplication du seul élément XML la contenant. Dans le premier cas, l'identifiant originel de l'élément peut être conservé mais le nom du document doit fatalement être changé. Dans le second cas, l'identifiant unique doit être modifié. Dans un cas comme dans l'autre, on rencontre de facto un problème avec le document qui contenait des liens vers cette donnée modifiée : dans le premier cas, il faut potentiellement changer tous les paragraphes contenant un quelconque lien vers le document révisé, et dans le second il faut seulement modifier les paragraphes contenant un lien vers la seule information modifiée. Toujours est-il que ces documents sont eux aussi modifiés, et, par souci de traçabilité, doivent encore être modifiés, et ainsi de suite !

Par effet de propagation, on aura compris que c'est potentiellement la totalité des documents qui doit être dupliquée en conséquence d'une unique modification de valeur dans un paragraphe d'un des documents de la base.

La solution

Pour lutter contre ce risque, la solution consiste à :

- ne pas chercher à multiplier les petites entités,
- mettre en place une politique d'adressage logique des ressources,

- étudier l'ensemble des mécanismes qui permettront de générer automatiquement le plus grand nombre possible de liens au moment de la publication de l'information.

Pour les cas les plus compliqués, le lecteur se reportera utilement au chapitre 13 traitant de la gestion des révisions, où est détaillé un modèle mettant en œuvre les URN (Uniform Resource Names).

Liens et gestion de configuration

La gestion de configuration consiste à gérer différentes versions d'un même matériel ou logiciel. La conséquence directe sur la documentation est qu'elle sera composée de parties communes, combinées avec des parties spécifiques.

L'impact sur les liens est immédiat. Quand les parties spécifiques viennent remplacer des parties communes, il faut garantir que tous les liens restent cohérents.

Pour illustrer notre propos, nous allons réutiliser l'exemple fourni en début de chapitre et illustré à la figure 12-3.

Rappelons au préalable un point évoqué à la section précédente : quand un fichier contient des données XML, il est possible de définir localement une variante en utilisant un balisage spécial, mais quand un fichier renferme des données non-XML, le seul moyen de créer une variante est d'en faire une copie. La gestion de configuration pose dès lors, au regard des liens, un problème sérieux.

Imaginons qu'un même appareil vendu en France, en Suisse et au Canada n'ait pas exactement la même configuration dans chacun de ces pays. À langue et structure logique égale, la structure d'assemblage de la documentation fera appel à des ressources physiques différentes.

Supposons par exemple que le module `mod1` contienne une illustration et qu'il existe deux versions du module `mod2` : l'une pour la France, l'autre pour la Suisse (`mod2-F.xml` et `mod2-S.xml`). L'illustration existe également en deux versions, l'une pour la France, l'autre pour la Suisse (`illus1-F.gif` et `illus1-S.gif`).

Le module `mod1.xml` est commun à toutes les versions :

```
<module>
<avcorps>
<linkId id="L1">
  <target id="e07652" type="element">i5</target>
</linkId>
<linkId id="F1">
  <target type="entity">l-illus1</target>
</linkId>
</avcorps>
```

```
<body>
<p id="p1">Ce paragraphe contient un lien vers un <link ref="L1">
élément</link> de document 2 et une <link ref="F1">illustration
</link>.</p>
</body>
</module>
```

Le module `mod2-F.xml` est spécifique à la France :

```
<module>
<avcorps>
<linkId id="L1">
  <target id="e07651" type="element">p1</target>
</linkId>
</avcorps>
<body>
<p id="i5">VERSION française. De ce paragraphe part un lien vers les
spécifications <link ref="L1">de lien e07651</link> local à ce document
2</p>
</body>
</module>
```

Le module `mod2-S.xml` est spécifique à la Suisse :

```
<module>
<avcorps>
<linkId id="L1">
  <target id="e07651" type="element">p1</target>
</linkId>
</avcorps>
<body>
<p id="i5">VERSION Suisse. De ce paragraphe part un lien vers les
spécifications <link ref="L1">de lien e07651</link> local à ce document
2</p>
</body>
</module>
```

Dès lors, on peut concevoir des documents XML faisant à la fois office de structure d'assemblage des publications et de base de transposition.

Ce qui donne pour la version française :

```
<pub version="France">
  <ressourceId id="L07651" type="entity">e07651</ressourceId>
```

```
  <ressourceId id="L07652" type="entity">e07652</ressourceId>
  <ressourceId id="l-illus1" type="entity">Illus1-F.gif</ressourceId>

 <section title="PCD" id=" i07650">
  <ressource title="FS Export"  url="file://c:/doc/mod1.xml"
             ressourceId="L07651"/>
 </section>
 <section title="sub-PCD" id=" i07652">
  <ressource title="FS Export" url="file://c:/doc/mod2-F.xml"
             ressourceId="L07652"/>
  </section>
</pub>
```

et pour la version suisse :

```
<pub version="Suisse">
  <ressourceId id="L07651" type="entity">e07651</ressourceId>
  <ressourceId id="L07652" type="entity">e07652</ressourceId>
  <ressourceId id="l-illus1" type="entity">Illus1-S.gif</ressourceId>
 <section title="PCD" id=" i07650">
  <ressource title="FS Export"  url=" file://c:/doc/mod1.xml"
             ressourceId="L07651"/>
 </section>
 <section title="sub-PCD" id=" i07652">
  <ressource title="FS Export" url=" file://c:/doc/mod2-S.xml"
             ressourceId="L07652"/>
 </section>
</pub>
```

Avec un tel modèle, le passage de la version française à la version suisse ne nécessite pas davantage que le changement de deux informations, ce qui est un gage important de cohérence du contenu final.

Ce type de modèle permet surtout d'aller encore plus loin et de produire automatiquement, à partir de règles génériques de gestion de configuration, les différentes versions de la structure d'assemblage.

Si le modèle et les éventuelles tables relationnelles sont relativement simples à mettre au point, il n'en est pas de même des interfaces utilisateurs de saisie : la manipulation de liens abstraits est quelque chose de difficile à appréhender pour l'esprit humain, et l'ergonomie des interfaces homme-machine est, dans le cas d'une saisie manuelle, difficile à obtenir de manière générique ; en revanche, il est possible de le faire pour des applications métier spécialisées.

Mécanismes permettant de contrôler les liens partir d'un schéma

Avant d'entrer dans le problème du contrôle de la pose des liens, rappelons les différents types de liens disponibles avec XML :

- Les liens hiérarchiques explicites issus de la nature même de XML. Il s'agit des relations de type parent-enfant et fratrie induites par la nature imbriquée du balisage XML.
- Les liens hiérarchiques implicites, rendus explicites par des expressions XPath (une table des matières par exemple).
- Les liens point à point exprimés par des couples ID/IDREF.
- Les liens identitaires de type Key/Keyref.
- Les liens autres exprimés par le modèle XLink.

Tous ces liens ne dépendent pas du schéma XML. Par exemple, les liens implicites créés par des expressions XPath (ou XQuery) échappent totalement au contrôle du schéma. Ces liens apparaissent postérieurement à la création du document XML et sont principalement le fruit des applications qui vont exploiter les données XML.

En revanche, les liens identitaires de type Key/Keyref dépendent exclusivement du schéma : seule sa connaissance permet de retrouver ces liens (le mécanisme des clés et références de clés permet, *via* un schéma, de garantir qu'une donnée est égale à une autre – ce qui lie implicitement les deux données entre elles).

C'est en évaluant les possibilités offertes par un schéma XML que l'on se fait une idée du contrôle qu'il exerce sur les liens. Par exemple, si un schéma XML n'autorise que le type ID/IDREF, on dira qu'il est fermé car il oblige les sources et les cibles des liens à être physiquement dans le même document. A contrario, un schéma XML mettant en œuvre le type `xs:anyURI` est ouvert car il permet d'utiliser n'importe quelle cible. Entre l'un et l'autre de ces deux extrêmes, il y a le mécanisme des facettes de XML Schema pour contrôler la forme des URI utilisées dans un document XML. Ainsi, il est possible de contrôler que le début d'une URI correspond à une liste de valeurs autorisées.

Le schéma XML suivant permet de contrôler que les URI utilisées dans l'élément `uri` respectent une forme lexicale commençant par `http://www.xmldev.com/`. Le type anyURI (repère ⓿) y est restreint par utilisation de la facette xs:pattern spécifiant une forme lexicale au moyen d'une expression régulière (repère ❶).

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:simpleType name="uriType">
  <xs:restriction base="xs:anyURI"> ← ⓿
   <xs:pattern value="http://www.xmldev.com/.*"/> ← ❶
  </xs:restriction>
 </xs:simpleType>
```

```
 <xs:element name="uris">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="uri" maxOccurs="unbounded">
     <xs:simpleType>
      <xs:list itemType="uriType"/>
     </xs:simpleType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Le premier fragment XML suivant est valide par rapport à ce schéma, tandis que le second ne l'est pas (les erreurs sont mises en gras) :

```
<?xml version="1.0" encoding="UTF-8"?>
<uris xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="7-facetteURI.xsd">
 <uri>http://www.xmldev.com/xmldata.xml</uri>
 <uri>http://www.xmldev.com/modele/exemple/index.html</uri>
</uris>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<uris xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="7-facetteURI.xsd">
 <uri>http://www.xmldev.net.org/xmldata.xml</uri>
 <uri>http://www.xmlbc.com/modele/exemple/index.html</uri>
</uris>
```

On peut aussi contrôler la forme lexicale des liens en utilisant une bibliothèque de valeurs référencées à l'aide des mécanismes `key/keyref` de XML Schema.

Le schéma suivant définit une bibliothèque d'URI tous différents (ce sont des clés de type `xs:key`) :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="uris">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="uri" maxOccurs="unbounded">
     <xs:simpleType>
      <xs:list itemType="xs:anyURI"/>
     </xs:simpleType>
    </xs:element>
```

```
    </xs:sequence>
   </xs:complexType>
   <xs:key name="uriDansBibliothèque">
    <xs:selector xpath="uri"/>
    <xs:field xpath="."/>
   </xs:key>
  </xs:element>
</xs:schema>
```

Ce schéma valide par exemple la bibliothèque d'URI suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<uris>
 <uri>http://www.xmldev.com/xmldata.xml</uri>
 <uri>http://www.xmldev.com/modele/exemple/index.html</uri>
</uris>
```

On peut dès lors écrire un schéma dans lequel certains éléments se verront imposer un contenu, lequel devra dans le cas présent être l'un des URI de la bibliothèque.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:include schemaLocation="7-bibUris.xsd"/>
 <xs:element name="collaborateurs">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="uris"/>
    <xs:element name="collaborateur" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="fiche" type="xs:anyURI"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
  <xs:key name="uriDeLaBibliothèque">
   <xs:selector xpath="uris/uri"/>
   <xs:field xpath="."/>
  </xs:key>
  <xs:keyref name="ficheCollaborateur" refer="uriDeLaBibliothèque">
   <xs:selector xpath="collaborateur/fiche"/>
   <xs:field xpath="."/>
  </xs:keyref>
```

```
  </xs:element>
</xs:schema>
```

Et voici une instance valide de ce schéma :

```
<collaborateurs>
<!--
<xi:include href="7-bibUris.xml" parse="text" encoding="CDATA"
            xmlns:xi="http://www.w3.org/2001/XInclude"/>
-->
<uris>
 <uri>http://www.xmldev.com/xmldata.xml</uri>
 <uri>http://www.xmldev.com/modele/exemple/index.html</uri>
</uris>

 <collaborateur>
  <fiche>http://www.xmldev.com/xmldata.xml</fiche>
 </collaborateur>
</collaborateurs>
```

REMARQUE Bibliothèque d'URI

Dans le dernier document XML, nous avons inséré en dur la bibliothèque d'URI, mais avons parallèlement mentionné l'inclusion qu'il faudrait déclarer avec `xi:include` de la recommandation XInclude du W3C pour éviter de devoir dupliquer la bibliothèque d'URI.

VOCABULAIRE Intégrité référentielle

L'intégrité référentielle d'un lien est sa cohérence : il s'agit non seulement de contrôler que la cible existe mais encore qu'elle est juste.

En conclusion, XML Schema ne permet pas à lui tout seul de bien contrôler l'intégrité référentielle des liens. On ne peut y procéder qu'au moyen de programmes de calcul capables de contrôler les liens au regard de règles métier.

Au mieux, un schéma contrôlera si la pose d'un lien est autorisée et si sa forme lexicale est correcte.

En résumé

Dans ce chapitre, nous avons fait un long périple au travers de différents modèles de liens et langages de liaison, depuis le cas des ID/IDREF, considéré comme le plus simple, à celui des cartes de topiques. Cependant, la dernière partie du chapitre a montré que des modèles particulièrement sophistiqués pouvaient être réalisés en s'appuyant sur le seul mécanisme des ID/IDREF (complété il est vrai d'un modèle relationnel) !

Est-ce à dire que le sujet des liens ne fait pas partie des plus simples ? La réponse est oui. Le sujet est complexe et complexes sont ses représentations.

L'affirmation « le volume crée le problème » est ici vérifiée. Créer un lien dans un document n'est pas un problème, mais en créer des centaines de milliers dans des milliers de documents est un problème qui nécessite une modélisation préalable. Nous avons offert dans ce chapitre une palette de solutions.

Si nous avons présenté en partie les vocabulaires normalisés des modèles Xlink et XTM, nous avons aussi insisté sur les problèmes de gestion que posent les liens. Réviser un document ou un ensemble d'informations, le transporter, et enfin en gérer l'applicabilité par rapport à une configuration de système est un problème complexe dès lors que des liens entrent en ligne de compte.

En particulier, la gestion combinée des liens, des révisions et des processus est une question délicate car, quand bien même on saurait résoudre le problème sur le plan technique, il n'en resterait pas moins qu'il faut offrir aux utilisateurs des interfaces conviviales de contrôle, d'accès et de saisie des liens ; ce n'est pas impossible, mais coûteux.

Nous allons étudier dans le chapitre suivant un autre problème potentiellement délicat, celui de la gestion des révisions.

13

Modèles pour la gestion des révisions et des versions

Voici une démarche, agrémentée d'exemples, qui permet de trouver le modèle approprié à la gestion de différentes révisions et versions de documents XML.

La gestion des révisions et des versions doit toujours, avec XML, être étudiée sous les deux facettes qui forment l'ossature de ce chapitre :

- celle du balisage à appliquer à l'intérieur du document XML,
- celle de la gestion de l'enveloppe, à savoir le document XML en tant que fichier.

Sur le plan de la modélisation XML, gérer des révisions ou des versions revient au même. Dans les deux cas, le problème est celui de l'identification des objets et de la commodité avec laquelle on peut passer de l'un à l'autre : d'une révision à une autre, d'une version à une autre.

Nous verrons dans ce chapitre des exemples et des modèles portant sur les deux cas de figure.

Gestion des révisions à l'intérieur d'un document XML

Introduction

La gestion des modifications apportées à un document XML est à la fois chose simple et compliquée :

- Simple, parce que la nature même du balisage XML permet d'envisager toutes sortes de marquage des révisions. Nous en rappelons l'essentiel dans cette introduction.
- Compliquée, car ce balisage soulève des problèmes spécifiques que nous détaillerons dans une section qui leur est dédiée.

Nous verrons que, sous certaines conditions, il est possible d'identifier les différences entre documents XML après y avoir porté des modifications.

Dès que l'on aborde l'étude du marquage des révisions, une première question vient à l'esprit : « Que doit-on indiquer au juste ? »

Effectivement, s'il semble évident de baliser tel ou tel paragraphe modifié et la version du document correspondante, il est ensuite beaucoup plus délicat de décider du type d'information de révision qu'il est souhaitable de conserver. Voici des exemples des informations susceptibles d'être conservées entre deux révisions :

- Qui a fait la modification ?
- Quelle est la nature de la modification ?
- Quelle est la raison de la modification ?
- Qu'est-ce qui, très exactement, a été modifié ?
- Qui a validé la modification ?
- Quelle est la date de la modification ?

Il y en a bien d'autres encore.

Les possibilités de balisage des révisions

Utilisation d'attributs

Il est possible de marquer les révisions en utilisant un ensemble d'attributs autorisés par la DTD ou le schéma sur tout ou partie des éléments XML.

Les attributs `revnum`, `revdate`, `revaut` et `revtype`, par exemple, enregistrent respectivement le numéro de version du document, la date de la révision, son auteur et son type.

Le balisage ci-après indique que le paragraphe a été modifié dans la version 2.1 du document, le 14 août 2004 à 9 h 30 par Didier :

```
<p revnum="2.1" revdate="2004-08-14T09:30" revaut="Didier"
   revtype="change">
```

Cette solution présente les avantages suivants :

- La structure originelle du document n'est pas modifiée. Les attributs sont ajoutés à tout moment, sur n'importe quel élément défini par une DTD ou schéma XML. Avec XML Schema, les attributs de révision peuvent même être ajoutés aux types en les dérivant par extension.
- Les valeurs utilisées seront partiellement contrôlées en utilisant des listes déroulantes de valeurs prédéfinies ou des dérivés de types simples en ce qui concerne XML Schema.

Cependant, elle a les inconvénients suivants :

- Pas de possibilité de borner précisément la portion modifiée du texte. Les bornes sont celles des éléments de la DTD ou du schéma qui, en général, n'ont pas été conçus dans une optique de gestion des révisions.
- Pas de possibilité de cumuler les révisions puisqu'on ne peut mettre par élément qu'un seul attribut de même nom. Avec cette technique, la recopie des fichiers entre deux révisions est obligatoire.

Utilisation de balises fixes

Nous qualifions ici de fixes les balises dont l'emplacement est prévu par la DTD ou le schéma XML, par opposition aux balises flottantes, spécificité de SGML qui sera présentée dans la prochaine section.

Les balises fixes peuvent être, au choix :

- trois éléments (par exemple `new`, `del` et `chg`) pour indiquer une nouveauté, une suppression ou un changement ;
- un seul élément (par exemple `rev`) accompagné d'un attribut spécifiant la nature de la modification (par exemple `change="new"`, `change="del"`, `change="mod"`).

La balise de fin délimite la zone révisée.

La balise de début peut porter d'autres attributs pour fournir des indications supplémentaires sur la révision.

Cette solution a l'avantage de rendre possible le cumul des révisions.

Toutefois, elle a les inconvénients suivants :

- Il est difficile d'introduire ces éléments dans la structure. S'il est simple de les introduire à l'intérieur des éléments textuels, il n'est pas aisé de le faire dans le cas de structures plus complexes comme des tableaux, chapitres, sections...
- En dehors des contenus textuels, cette technique apporte plus d'inconvénients que d'avantages par rapport à la technique des attributs vue à la section précédente.

Le balisage suivant a été produit par Microsoft Word™ 2000 en mode révision. Nous l'avons simplifié et mis en gras pour le rendre explicite :

```
<p><del cite="Didier" datetime="2004-08-14T09:30">toto</del>
<ins cite="Didier" datetime="2004-08-14T09:30">jjt</ins>
</p>
<table>
<tr style='mso-table-inserted:Didier 20040814T0933'>
  <td>
  <p><ins cite="Didier" datetime="2004-08-14T09:33">a</ins></p>
  </td>
```

On observe que Microsoft Word™ utilise la technique du balisage quand il s'agit de marquer les révisions du contenu textuel, mais celle des attributs pour marquer la révision des structures complexes, ici un tableau.

Il est à noter que Word utilise dans le cas de l'élément `tr` un attribut (`style`) suffixant un attribut standard et non des attributs spécifiquement dédiés à la gestion des révisions. On pourra noter également que l'indication d'insertion du tableau (`mso-table-inserted`) a été enregistrée sur la première rangée du tableau et non sur l'élément `table` lui-même. Comme nous l'avons dit, la technique qui consiste à utiliser les attributs ne permet pas le cumul des révisions. Aussi, comme vous l'aurez peut-être déjà remarqué, Word informe les utilisateurs que les modifications sur les tableaux (notamment les suppressions) ne peuvent pas être gérées en révision.

On se doute que ce type de balisage est quasi impossible à poser à la main. Sa génération doit être assurée par l'éditeur XML utilisé.

Utilisation de balises flottantes

L'utilisation de balises flottantes est une spécificité de SGML. Pour fonctionner, cette technique s'appuie sur les grammaires de Relax NG et le mécanisme des exceptions qui n'existe ni dans les DTD de XML 1.0 ni dans les schémas de XML Schema.

Le mécanisme des exceptions consiste à autoriser ou interdire l'usage d'éléments spécifiés à partir d'un certain élément de l'arbre. Ainsi, il est possible d'indiquer dans une

DTD qu'une balise, `revst` par exemple, est autorisée n'importe où à partir de l'élément racine. Il suffit pour cela d'écrire `+(revst)` comme dans l'exemple ci-après :

```
<!ELEMENT root - - (title,para+) +(revst)>
```

Une révision chevauche souvent plusieurs éléments. Aussi, pour satisfaire à l'idée qu'une révision commence dans un élément et se termine dans un autre, deux balises indépendantes et mises en exception sont autorisées :

- une balise de début de révision, `revst` par exemple ;
- une balise de fin de révision, `revend` par exemple.

Dans la DTD, cela se traduit par la définition suivante :

```
<!ELEMENT root - - (title,para+) +(revst|revend)>
```

Ainsi, il est possible de baliser une modification qui figurerait à cheval sur deux paragraphes :

```
<PARA>aaaaaaaa<REVST/>ffffffffffffff</PARA>
<PARA>fffffffff<REVEND/>aaaaaaaaa</PARA>
```

Malheureusement, il est tout autant valide d'écrire :

```
<PARA>aaaaaaaa<REVEND/>ffffffffffffff</PARA>
<PARA>fffffffff<REVST/>aaaaaaaaa</PARA>
```

En effet, le mécanisme des exceptions ne contrôle pas l'ordre des éléments ainsi utilisés.

Cette solution présente les avantages suivants :

- La facilité de mise en œuvre : les marques de révisions peuvent chevaucher n'importe quel élément de structure.
- Les marques de révision peuvent être porteuses de leurs propres attributs de qualification. La marque de fin peut être porteuse d'attributs tout autant que la marque de début.
- Permet de cumuler les révisions.
- Très facile à introduire dans n'importe quelle DTD SGML.

Cependant, elle a les inconvénients suivants :

- Elle ne permet pas de garantir la parité des balises de début et de fin. Il s'agit de deux éléments vides, indépendants l'un de l'autre. Nous avons vu que de grossières erreurs de balisage pouvaient être commises sans que le parseur s'en aperçoive.

- Elle donne quelquefois des informations contradictoires avec la structure. Dans l'exemple suivant, l'ordre de destruction porte sur l'élément de début de paragraphe sans recouvrir la balise fermante. Le fragment que nous donnons en exemple est pourtant structurellement juste : aucun parseur ne verra l'aberration de la situation jusqu'au moment où la portion détruite sera physiquement supprimée du document XML. À ce moment là seulement, apparaîtra un sérieux problème de structure.

```
<DOCTYPE root [
<!ELEMENT root - - (para+) +(revst|revend)>
<!ELEMENT para - O (#PCDATA)>
<!ELEMENT (revst|revend) - O EMPTY>
<!ATTLIST(revst,revend) revtype (del|new|chg) 'ins'
revindicator id #REQUIRED>
>
<root>
<REVST revtype="del" revindicator="a412"/><PARA>aaaaaaaa<REVEND
       revtype="del" revindicator="a412"/>ffffffffffffff</PARA>
</root>
```

Ces quelques exemples disent assez à quelle difficulté on s'expose avec ce type de balisage que nous devons étudier en détail.

Les difficultés du balisage des révisions

Balisage contraint par le balisage principal

Nous qualifions ici de balisage principal celui qui structure véritablement le document XML ; par opposition au balisage qui pourrait être appelé secondaire, car non déterminant pour la structure du document (par exemple le balisage dédié aux métadonnées, aux mises en valeur, aux révisions le cas échéant...).

Ce qui est intéressant, c'est qu'en faisant porter les informations de révision par le balisage principal, on met l'accent sur les modifications apportées à la structure principale du document.

Certes, la précision en souffre puisque, comme nous l'avons vu plus haut, il est impossible d'encadrer le mot –voire le caractère – modifié.

Nous allons hélas soulever ici un problème profondément antinomique avec le concept même de balisage principal. Ce problème survient à partir du moment où l'on souhaite garder l'historique des révisions.

Envisageons la DTD XML suivante, somme toute très banale :

```
<DOCTYPE root [
<!ELEMENT root (section+)>
<!ELEMENT section (titre,para+)>
<!ELEMENT titre (#PCDATA)>
<!ELEMENT para (#PCDATA)>
<!ATTLIST titre revnum CDATA #REQUIRED
revtype (new|del|ch) 'ins'>
<!ATTLIST para revnum CDATA #REQUIRED
revtype (new|del|ch) 'ins'>
]>
```

Elle valide le document suivant :

```
<root>
 <section>
  <titre revnum="1.0" revtype="new">Ceci est le titre originel</titre>
  <para>aaaaaaaa</para>
 </section>
</root>
```

Et, si le titre est modifié et que l'on souhaite garder l'historique, cela devrait donner en version 1 :

```
<root>
 <section>
  <titre revnum="1.0" revtype="new">Ceci est le titre originel</titre>
  <titre revnum="2.0" revtype="chg">Ceci est le titre modifié</titre>
  <para>aaaaaaaa</para>
 </section>
</root>
```

Or, la DTD interdit de dupliquer la balise `titre`. Cette version du document n'est donc pas valide.

En extrapolant ce problème, on s'aperçoit que, pour garder l'historique des révisions, il devrait être possible de répéter tous les éléments de la DTD. Ce qui est, bien évidemment, totalement antinomique avec le concept de balisage principal. Voilà pourquoi la gestion des révisions devient rapidement incompatible avec la simple pose d'attributs sur le balisage principal.

Toutefois, lorsque cette technique est retenue, la seule solution pour garder l'historique consiste à dupliquer le document XML à chaque nouvelle révision.

Révision des attributs

Les attributs des documents XML peuvent contenir des informations importantes. La gestion des révisions peut également s'appliquer aux attributs.

Or, il n'est pas possible de conserver les valeurs successives d'un attribut. Le seul moyen d'y pourvoir, c'est de dupliquer l'élément auquel il appartient.

Quand la gestion des révisions s'impose pour les attributs, il convient de réfléchir à la possibilité de remplacer les attributs concernés par des éléments simples.

Les liens

Comme nous l'avons exprimé au chapitre précédent, la gestion des révisions peut avoir des conséquences non négligeables quand des liens sont en jeu.

L'univers des documents XML est composé de fichiers contenant des données balisées et des entités graphiques, le plus souvent de format binaire.

En ce qui concerne les entités graphiques, il est aisé de comprendre que toute gestion des révisions implique de procéder à une duplication du fichier graphique avant de le modifier.

De leur côté, les documents XML doivent également être dupliqués car il advient toujours un moment où il n'est plus possible de cumuler les modifications par un simple balisage interne (voir section précédente). Eux aussi doivent être dupliqués.

Dans les deux cas, le problème qui se pose est celui d'une duplication des fichiers, et donc des liens qui doivent s'adapter à leur nouvelle cible.

Avec les documents XML, le problème de la gestion des révisions prend une nouvelle dimension, les liens et la modularisation étant désormais faciles à réaliser.

Nous verrons ci-après le cas concret de la norme S1000D : cette norme généralise l'usage des noms uniformes de ressources (URN, pour *Uniform Ressource Name*) et de serveurs d'URN comme réponse à ce problème délicat.

Difficultés pour les auteurs

On s'en sera douté au vu des balisages présentés dans la première section de ce chapitre, placer les informations de révision n'est pas chose simple pour les auteurs.

Tout d'abord, ces derniers doivent initialiser les informations de révision puis, à chaque modification, ne pas oublier de renseigner les champs de gestion correspondants. C'est évidemment une tâche trop complexe, risquant d'entraîner de nombreuses erreurs.

Si l'éditeur XML utilisé n'est pas capable de poser automatiquement les marques de révision, il convient de développer des programmes de comparaison. Ainsi, les

auteurs peuvent librement modifier un document XML qui sera ensuite comparé à sa version précédente par un programme.

Pour que la comparaison soit efficace et sans ambiguïté, les différentes versions des documents XML doivent avoir la même forme canonique. La définition d'une forme canonique fait l'objet de la recommandation du W3C « Canonical XML » du 15 mars 2001. On désigne par là le format d'enregistrement d'un document XML dans lequel toutes les valeurs par défaut des attributs ont été mises, les blancs nettoyés, les entités appelées remplacées par le texte littéral qu'elles représentent, les commentaires supprimés, les formes lexicales des types de base rétablies dans leur forme officielle, etc.

On le voit, la comparaison entre deux versions de documents XML, *a posteriori* de leur saisie, oblige à les canoniser, ce qui a pour effet d'en modifier quelque peu l'aspect. Cela n'est, la plupart du temps, qu'un épiphénomène.

Un exemple de balisage

Le modèle que nous présentons ici est celui de l'ATA2100, un standard élaboré pour la documentation technique des avions et de leurs équipements. Cette norme concerne aussi bien l'ensemble de l'avion que des petits équipements.

Pour des raisons de sécurité, la gestion des révisions est un point important de la documentation des avions. La traçabilité des modifications doit être assurée par plusieurs intervenants et plus d'une dizaine de types différents de documents, tous utilisant les principes de gestion des révisions que nous présentons dans cette section.

Structure racine

La racine des documents techniques met immédiatement l'accent sur la gestion des révisions. Dans la figure 13-1, où nous avons pris pour exemple la DTD d'un manuel de maintenance, les éléments `increv` et `tr` sont notés respectivement pour *incremental revision* et *temporary revision*. Ces deux éléments et la séquence (`title`, `mfmatr`, `chapter`) s'excluent mutuellement. Cela signifie qu'une mise à jour ne concerne toujours qu'une partie du document originel (représenté par la séquence `title`, `mfmatr`, `chapter`). Le principe retenu par les concepteurs de ce modèle est celui d'une publication initiale, comportant ensuite des mises à jour incrémentales.

Deux types de mises à jour sont pris en compte : les modifications temporaires avec l'élément `tr`, les mises à jours définitives avec l'élément `increv`.

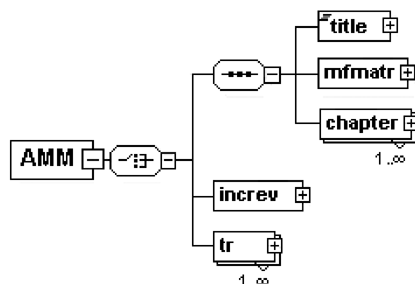


Figure 13–1
La structure supérieure d'un manuel de maintenance

L'élément racine `amm` porte les quelques attributs de révision qui s'appliquent à l'ensemble du document : la date de première publication du manuel (`oidate`), la date de la dernière révision (`revdate`), son état (`chg`). En voici un exemple :

```
<amm model="747" docnbr="747-21-2360" spl="81205" tsn="1"
oidate="19940707" revdate="19950511" chg="R" lang="EN">
```

Les sections suivantes vont présenter les mécanismes internes à la structure mise en œuvre pour la gestion des révisions.

Structure de l'élément mfmatr

L'élément `mfmatr` est l'avant-corps du manuel. Il est constitué de plusieurs parties concernant pour la plupart la gestion des révisions et l'applicabilité du manuel. Dans cet avant-corps apparaît la liste des révisions temporaires du document : `trlist` (littéralement *temporary revision list*). Cette liste correspond aux modifications publiées au moyen de l'élément `tr` expliqué plus loin.

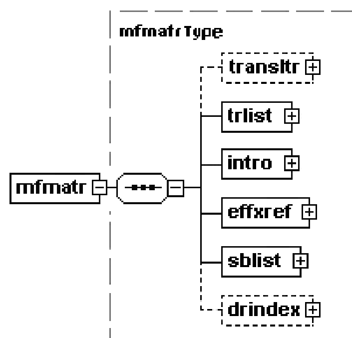


Figure 13–2
Structure de l'élément mfmatr

Structure de l'élément chapter

Dans notre cas de figure, un *chapitre* est une entité répétable qui forme le corps du document technique.

Deux éléments concernent la gestion des révisions (`chgdesc` comme *change description* et `deleted`, utilisé au cas où le chapitre serait entièrement retiré).

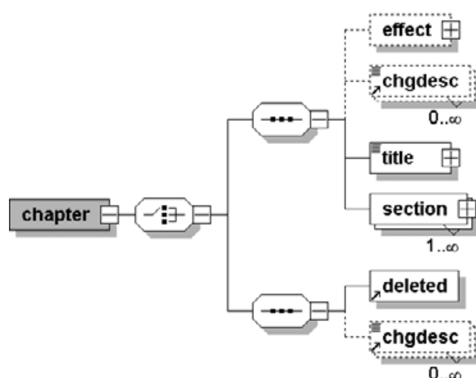


Figure 13–3
Structure de l'élément chapter

Structure de l'élément increv

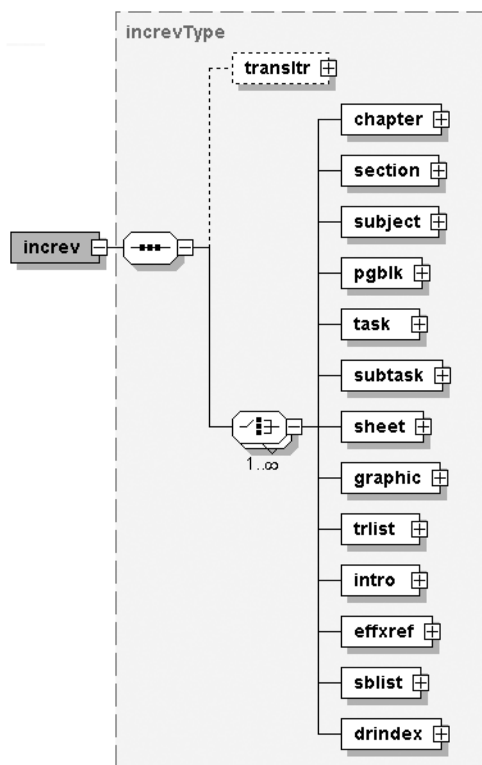
Le nom `increv` est l'abréviation de *incremental revision*, soit *révision incrémentale*. Il s'agit d'un mécanisme de base isolant du reste du document les unités textuelles mises à jour. Treize nœuds, ou ancres – car on parle à leur sujet de points *d'ancrage* des révisions –, sont autorisés à prendre en charge cette opération. La figure 13-4 en montre la liste.

La structure de l'élément `increv` possède une particularité : tous les éléments sont au même niveau hiérarchique alors qu'ils correspondent à des niveaux de profondeur bien différents dans la structure originelle du document. L'élément `section` figure en temps normal sous l'élément `chapter`. Comme nous le disions en introduction de cet exemple, les concepteurs du modèle ont prévu que les parties révisées pourraient faire l'objet d'une publication indépendamment du reste du document.

Chaque ancre porte trois attributs de révision :

- `chg`, comme *change*, peut prendre les valeurs `N`, `D`, `R` et `U`, qui signifient respectivement *New*, *Deleted*, *Revised* et *Unchanged*.
- `key`, comme clé, est un identifiant unique de type ID construit avec des règles très précises permettant d'assurer la traçabilité des révisions.
- `revdate` contient la date de la révision.

Figure 13–4
Structure de l'élément increv montrant les treize nœuds d'ancrage des révisions



À l'intérieur d'une ancre, les zones réellement modifiées sont encadrées par les balises de révision `revst` et `revend`.

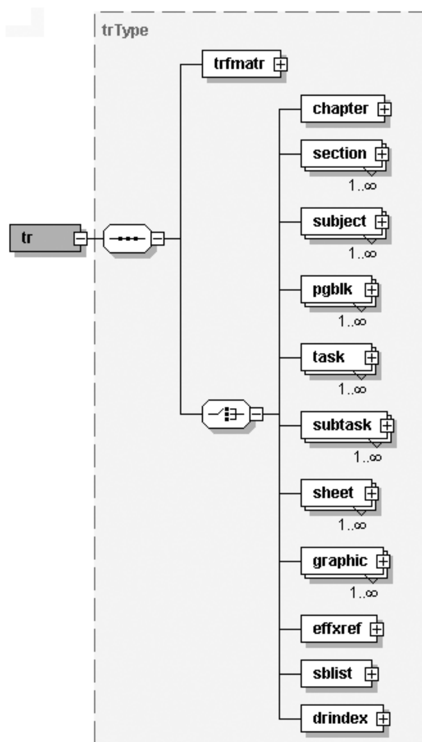
On remarquera que l'élément `trlist` (liste des révisions temporaires) fait lui-même l'objet d'un processus de révision incrémentale puisqu'il est dans la liste des ancres : en effet, les blocs d'information dédiés à la gestion des révisions sont eux-mêmes révisés et sont donc soumis aux mêmes règles de gestion des révisions que les autres éléments textuels ou graphiques.

Structure de l'élément tr

Les révisions temporaires correspondent au concept de *page bis* ou *pages jaunes*. Les pages sont insérées dans les publications en complément des anciennes pages.

Le modèle de contenu de l'élément `tr` est, à peu de chose près, le même que celui de `increv`, et les mêmes ancres peuvent faire l'objet soit d'une révision temporaire, soit d'une révision incrémentale, à l'exception de l'élément `intro` et de la liste des révisions temporaires (`trlist`) qui ne fait pas elle-même l'objet d'une mise à jour temporaire.

Figure 13–5
Structure de l'élément tr dédié aux révisions temporaires



Ces structures montrent comment les étages supérieurs du modèle structurent la gestion des révisions. Nous allons désormais montrer ce qui se passe au niveau des paragraphes.

Le bornage précis des révisions

Les modifications sont marquées dans le corps du texte par deux éléments flottants, dénommés `revst` et `revend`. Concernant la technique que mettent en œuvre ces éléments *flottants*, nous renvoyons au début de ce chapitre.

Ces éléments n'ont comme durée de vie qu'une seule révision et sont réinitialisés d'une révision à l'autre. Ils ont comme objectif de marquer les différences d'une révision *n* à *n + 1*.

Ce qui était indiqué comme révisé (attribut `chg="R"`) à une révision donnée deviendra `U` *Unchanged* à la révision suivante sauf si, bien sûr, le contenu de l'ancre porteuse de cette information venait à changer à nouveau.

Ces règles vont être illustrées par des cas typiques. Nous allons tout d'abord utiliser un fragment simple composé d'un chapitre, d'une section et d'une sous-section. Au départ, l'indicateur de révisions vaut `U` comme *unchanged*.

```
<AMM>
<CHAPTER CHG="U" >
  <TITLE>aaaaaaaaaaaaaaaaaaaaa</TITLE>
  <SECTION CHG="U">
   <PARA>bbbbbbbbbbbbbbbbbbbbb</PARA>
   <SUBSEC CHG="U">
    <PARA>ddddddddddddddddddddd</PARA>
   </SUBSEC>
  </SECTION>
</CHAPTER>
</AMM>
```

Quand le premier paragraphe de la section est modifié, l'indicateur de révision de cette section, et uniquement de celle-là, passe à `R` comme *revised*. Le chapitre n'est pas pour autant indiqué comme ayant été révisé ; la modification effectuée est circonscrite à la `section`, et le chapitre reste à l'état `U`.

```
<AMM>
<CHAPTER CHG="U">
  <TITLE>aaaaaaaaaaaaaaaaaaaaa</TITLE>
  <SECTION CHG="R" revdate="19941201" key="M1">
   <PARA><REVST>ffffffffffffffffffffffff<REVEND></PARA>
   <SUBSEC CHG="U">
    <PARA>dddddddddddddddddddd</PARA>
   </SUBSEC>
  </SECTION>
</CHAPTER>
</AMM>
```

Lors de l'édition suivante, une information directement dépendante du chapitre est modifiée, et le chapitre porte alors l'information de révision. L'indicateur de révision de la section précédemment modifiée revient à `U` puisque ces indicateurs sont réinitialisés à chaque nouvelle révision.

```
<AMM>
<CHAPTER CHG="R" revdate="19951201" key="M2">
  <TITLE><REVST>revised title<REVEND></TITLE>
  <SECTION CHG="U">
   <PARA>ffffffffffffffffffffffffff</PARA>
```

```
    <SUBSEC CHG="U">
     <PARA>ddddddddddddddddddddddddddd</PARA>
    </SUBSEC>
  </SECTION>
</CHAPTER>
</AMM>
```

La destruction du contenu d'une ancre n'est possible que si la suppression de ce contenu est autorisée par le modèle. Elle est alors remplacée par le seul élément vide `deleted` et le fragment balisé devient :

```
<AMM>
<CHAPTER CHG="U">
  <TITLE>revised title</TITLE>
  <SECTION chg="D" revdate="19961201" key="M3">
    <REVST>
    <DELETED>
    <REVEND>
  </SECTION>
</CHAPTER>
</AMM>
```

Quand la destruction totale du contenu n'est pas possible ou quand la destruction ne concerne qu'une partie du texte, le balisage prend la forme suivante :

```
<AMM>
<CHAPTER CHG="U">
  <TITLE>revised title</TITLE>
  <SECTION CHG="U">
    <PARA>fffffffffffffffffffffffffff</PARA>
    <SUBSEC chg="R" revdate="19971201" key="M4">
     <PARA><REVST><REVEND></PARA>
    </SUBSEC>
  </SECTION>
</CHAPTER>
</AMM>
```

dans laquelle les balises `<revst>` et `<revend>` encadrent un contenu vide.

Pour compléter la gestion des révisions au niveau individuel, une liste récapitulative des nœuds modifiés est élaborée. Cette *liste des ancres effectives* est aux révisions ce qu'une table des matières est aux chapitres d'un ouvrage. Elle est introduite par l'élément `LEA` (acronyme de Liste of Effective Anchors) et est composée d'éléments dont le nom est `ANCHOR`. On y remarque les attributs `key` et `treeloc` : le premier contient

les identifiants des ancres, et le second la position nodale de chacune d'entre elles. Il s'agit d'une double identification qui, de ce fait, est redondante.

```
<LEA SPL="ATA/AIA" MODEL="A320" OIDATE="19920101" REVDATE="19920401"
     TSN="1" CUS="XXX" LANG="EN" DOCID="DOC">
<ANCHOR KEY="A112" REVDATE="19920101" CHG="U" TAGNAME="CHAPTER"
        TREELOC="2">
<ANCHOR KEY="A113" REVDATE="19920401" CHG="R" TAGNAME="SECTION"
        TREELOC="3">
<ANCHOR KEY="A232" REVDATE="19920101" CHG="U" TAGNAME="CHAPTER"
        TREELOC="14">
<ANCHOR KEY="A233" REVDATE="19920101" CHG="U" TAGNAME="SECTION"
        TREELOC="15">
<ANCHOR KEY="A235" REVDATE="19920101" CHG="U" TAGNAME="SECTION"
        TREELOC="21">
<ANCHOR KEY="A341" REVDATE="19920101" CHG="U" TAGNAME="CHAPTER"
        TREELOC="26">
<ANCHOR KEY="A342" REVDATE="19920101" CHG="U" TAGNAME="SECTION"
        TREELOC="27">
<ANCHOR KEY="A343" REVDATE="19920101" CHG="U" TAGNAME="SECTION"
        TREELOC="31">
</LEA>
```

La valeur de l'attribut `treeloc` est le numéro d'ordre de l'ancre par rapport à la racine du document. Ce dernier obéit à des règles très précises de comptage des balises afin que, d'une révision à l'autre, un même nombre représente une même position dans l'arbre XML.

C'est sur cette dernière considération que nous quittons la présentation du modèle de gestion des révisions utilisé dans la documentation technique de l'aviation civile.

Exemple basé sur l'utilisation des URN

Dans le modèle S1000D, de type modulaire, on procède à la gestion des révisions au travers d'indicateurs placés sur quelques éléments choisis : ceux qui constituent le corps textuel du module et dont le modèle de contenu n'est pas mixte. C'est le cas des titres, lignes de tableaux, alinéas (un niveau de liste donné), figures complètes, paragraphes seuls, etc. La marque de révision est posée *via* l'attribut `change` qui prend les valeurs suivantes :

- `add` si la balise est nouvelle par rapport à la version précédente du module,
- `mod` si un attribut ou le contenu de la balise a été modifié,

- `del` si une balise a été supprimée. L'attribut `change` est alors porté par le parent de celle qui a été supprimée.

Il a été décidé que les identifiants des ressources qui sont des cibles potentielles de liens resteraient constants même en cas d'évolution de leur contenu, cela afin d'éviter les problèmes de propagation aux liens des changements d'indice de révision des modules de données (problème exposé au chapitre 6). Ainsi, les identifiants des modules (le DMC), publication (le TPC) et illustration (l'ICN) restent inchangés tout au long de la vie de ces objets, et tous les liens les référençant sont constamment valides.

Avec l'avènement du Web, le mécanisme d'adressage par les URL (Uniform Resource Locators) a permis de définir une méthode d'accès à des ressources situées n'importe où géographiquement.

Hélas, les URL établissent des liens directs vers une localisation physique donnée, et les conséquences en sont importantes. En effet, si la ressource change de place, le lien est perdu, et cela donne lieu au message d'erreur standard 404 précisant que la cible n'a pas été retrouvée. De même, pour positionner un lien en utilisant les URL, la localisation physique de la cible doit être connue à l'avance, ce qui n'est pas toujours le cas.

Des solutions ont été élaborées pour définir des mécanismes de liaisons indépendants de la localisation physique des ressources. Ceux de la S1000D font intervenir les Uniform Resource Names (URN), ou *noms uniformes de ressources*.

Introduction aux URN

Par le biais de l'IETF, la communauté Internet a depuis longtemps exprimé le besoin de disposer d'un schéma d'adressage logique des ressources. Il en est résulté en 1997 la RFC 2141 qui normalise les noms uniformes de ressources. Avec les URN, les ressources sont identifiées par une clé invariante, ensuite résolue par un serveur de localisation capable de retrouver la localisation physique réelle de la ressource.

Bien que cet adressage indépendant réponde sur le principe au besoin, son développement en tant que méthode standardisée pour la résolution de la localisation des ressources s'est néanmoins fait attendre, et ce en grande partie en raison de l'incompatibilité technique des solutions développées : Persistent URL ou PURL, Basic URN Service résolution ou BURNS, ou encore Trivial HTTP ou THTTP.

PURL est une méthode de catalogage des ressources Internet développée par l'OCLC, un service de catalogage de librairies Internet qui repose lui-même sur un service web capable de retourner une adresse physique à partir d'une adresse logique. Cette méthode ne répond malheureusement pas vraiment au problème posé par la résolution des liens dans une documentation modulaire.

BURNS est un serveur HTTP d'URN résolus reposant sur une base de données X.500 développée par le DTSC de l'Université de Queensland en Australie. L'un des intérêts de la méthode est l'utilisation des URC, ou Uniform Resources Characteristics, pour récupérer les métadonnées de type Dublin Core (voir chapitre 11) associées aux ressources.

Nous allons maintenant donner quelques explications techniques complémentaires, notamment des définitions, relativement aux URN.

Ressources, URN, URI et URL

L'un des fondements du World Wide Web est la notion de ressource. Une ressource est une entité informatique quelconque adressée via un identifiant. Ce dernier peut être un nom logique (il s'agit alors d'un URN) ou une localisation physique (une adresse URL). Ces deux types d'adressage ont des syntaxes qui sont des variantes d'une même syntaxe générique : celle des identifiants uniformes de ressources du Web ou URI (Uniform Ressource Identifier).

La syntaxe des URI est spécifiée par la RFC IETF 2396 d'août 1998. Les syntaxes particulières des URL et des URN sont respectivement spécifiées par les RFC 1738 de décembre 1994, et 2141 de mai 1997.

Serveur de ressources

La résolution des ressources requiert que soit déterminée leur localisation physique à partir d'un URI. Un serveur de ressources a ainsi pour fonction de reconnaître la syntaxe de l'URI reçu, de déterminer s'il s'agit d'un URN ou d'une URL. Dans le cas d'une URL, le serveur donne directement accès à la ressource, tandis que dans celui d'un URN il doit activer le service capable d'en fournir l'URL.

Métadonnées

La gestion des accès aux ressources via des adresses logiques n'a de sens que si le serveur de ressources dispose d'un minimum d'intelligence pour les retrouver, par exemple pour retrouver la bonne version de la ressource à utiliser. C'est ici que les métadonnées de gestion associées aux ressources prennent toute leur importance.

Nous avons vu au chapitre 12 le modèle de métadonnées qui est utilisé par la norme S1000D.

Le serveur de ressources est un programme qui doit être capable de résoudre les URN en exécutant des recherches sur les métadonnées des ressources.

Format des URN

Le format des URN répond à la syntaxe `URN:NID:NSS`, dans laquelle :

- `URN` : est un préfixe obligatoire.

- `NID` : est l'identifiant d'un espace de noms (Namespace IDentifier).
- `NSS` : est la chaîne d'identification de la ressource à l'intérieur de l'espace de noms (Namespace Specific String).

Pour la S1000D, l'identifiant de l'espace de noms des ressources, le `NID`, est naturellement le mot `S1000D`.

Pour la partie `NSS`, les codes des modules de données (ou DMC comme Data Module Code), des publications (ou TPC comme Technical Publication Code) et enfin des illustrations (ou ICN comme Illustration Code Number) sont utilisés ; ce qui, au final, donne les trois possibilités de codification suivantes pour la S1000D :

```
URN:S1000D:DMC
URN:S1000D:TPC
URN:S1000D:ICN
```

Bien que les systèmes de codification des DMC, TPC et ICN soient semblables, un serveur de ressources doit savoir les différencier. Le principe visant à utiliser les mots DMC, TPC et ICN comme préfixes de la partie `NSS` a aussi été retenu. Les URN sont dès lors structurés ainsi :

```
URN:S1000D:DMC-{DMC selon la syntaxe S1000D}
URN:S1000D:TPC-{TPC selon la syntaxe S1000D}
URN:S1000D:ICN-{ICN selon la syntaxe S1000D}
```

Par exemple, voici l'URN correspondant au DMC `AE-A-00-40-05-50A-000A-A` :

```
URN:S1000D:DMC-AE-A-00-40-05-50A-000A-A
```

Enfin, cette syntaxe est complétée au besoin d'un numéro de révision et d'un code langue identifiés respectivement par les chaînes `_I-{ISSUE}` et `_L-{LANG}` ou `{ISSUE}` et `{LANG}`.

Par exemple, voici l'URN de la version 2 anglaise du module dont le DMC est `AE-A-00-40-05-50A-000A-A` :

```
URN:S1000D:DMC-AE-A-00-40-05-50A-000A-A_I-2_L-EN
```

Liens réalisés au moyen des URN

Dans la S1000D, les liens sont spécifiés en utilisant le vocabulaire de Xlink.

Dans ce premier exemple, le lien est une référence à un module :

```
<refdm xlink:type="simple"
  xlink:actuate="onRequest"
  xlink:show="replace"
  xlink:title="IETP-X Resolution de Ressource"
  xlink:href="URN:S1000D:DMC-AE-A-00-40-50-50A-000A-A">
<avee>
  <modelic>AE</modelic>
  <sdc>00</sdc>
  <chapnum>00</chapnum>
  <section>4</section>
  <subsect>0</subsect>
  <subject>05</subject>
  <discode>50</discode>
  <discodev>A</discodev>
  <incode>000</incode>
  <incodev>A</incodev>
  <itemloc>A</itemloc>
</avee>
</refdm>
```

Dans ce deuxième exemple, le lien est une référence à une publication :

```
<reftp xlink:type="simple"
  xlink:actuate="onRequest"
  xlink:show="replace"
  xlink:title="Exemple IETP-X"
  xlink:href="URN:S1000D:TPC-1B-B-AMP-27-I">TPC-1B-B-AMP-27-I</reftp>
```

Dans ce troisième exemple, le lien est une référence à un graphique :

```
<figure id="fig001">
  <title>Visualisation tete basse</title>
  <graphic xlink:type="simple"
    xlink:actuate="onLoad"
    xlink:show="embed"
    xlink:href="URN:S1000D:ICN-1A-B-311501-B-F6117-00352-A-01-1"
    boardno="ICN-1A-B-311501-B-F6117-00352-A-01-1"/>
</figure>
```

Pour convertir un URN en nom de fichier adressable, une méthode de résolution simple consiste à récupérer la chaîne correspondant à la valeur NSS de l'URN, à lui ajouter l'extension appropriée (XML, CGM, JPG…), et éventuellement à ajouter

une structure statique de répertoires afin de créer une URL relative ou statique. Une application plus complexe utilisera une base de données pour retrouver l'URL correspondant à un URN spécifique.

Service Web pour résoudre les URN

Pour être transformés en URL de manière standard et uniforme, les URN sont envoyés à un serveur de ressources qui active un programme de transcodage. Il s'agit typiquement d'un service Web acceptant des requêtes du genre :

```
http://urn.server.com/urnServiceName?urnQueryString
```

où :

- `urn.server.com` est le DNS du serveur de résolution.
- `urnService` est le nom du service de résolution d'URN.
- `urnQueryString` est la requête de transcodage.

Un tel service web n'est pas limité au transcodage des URN en URL mais répond également aux requêtes d'interrogation sur les métadonnées : soit parce qu'il possède les méthodes d'accès à la base de données qui les gère, soit parce que, connaissant les adresses physiques des ressources, il est capable d'aller y puiser les valeurs des métadonnées demandées.

Quand un couple nom/valeur est spécifié, cela signifie que la requête est qualifiée par des précisions quant à la valeur des métadonnées. Dans l'exemple 1 ci-après, on demande l'URL de la version anglaise d'une ressource.

Exemple 1 : pour retrouver l'URL en version anglaise d'un URN :

```
http://urn.server.com/urnService?urn=URN:NID:NSS&lang=EN
```

Quand un nom de caractéristique est transmis au service web sans valeur associée, cela signifie que la requête interroge le service web sur cette caractéristique. Dans l'exemple 2 ci-après, on cherche à obtenir le nom d'un auteur, et dans l'exemple 3 on s'enquiert simplement de l'URL d'une ressource.

Exemple 2 : pour retrouver la valeur de la caractéristique `auteur` de la version anglaise d'une ressource :

```
http://urn.server.com/urnService?uri=URN:NID:NSS&author&lang=EN
```

Exemple 3 : retourne l'URL de la version anglaise d'une ressource :

```
http://urn.server.com/urnService?urn=URN:NID:NSS&url&lang=EN
```

Le mot-clé `allurcs`, comme *all URC* (Uniform Ressource Characteristics), signifie que l'on demande au service toutes les métadonnées de la ressource spécifiée, comme dans l'exemple 4 :

Exemple 4 : pour retrouver la valeur de toutes les caractéristiques d'une ressource :

```
http://urn.server.com/urnService?uri=URN:NID:NSS&allurcs
```

Exemple 5 : requête permettant de savoir si deux URN sont équivalentes :

```
http://urn.server.com/urnService?urn=URN:NID:NSS1&urn=URN:NID:NSS2
```

On comprendra que, avec cette approche, le serveur d'URN est proche des fonctions de GED (Gestion électronique des documents).

Exemple concret

Le tableau suivant contient une série d'équivalences URN/URL, complétées des métadonnées `titre` et `langue`. La colonne « Défaut » indique l'URL considérée par défaut pour les requêtes sans précision sur un URN auquel sont associées plusieurs URL.

Tableau 13–1 Exemple de données

URN	URL	Titre	Langue	Défaut
URN:S1000D:DMC-999-A	www.ex.com/ ➥DMC-999-A.XML	Title 1	EN	O
URN:S1000D:DMC-520-A	www.ex.com/ ➥DMC-520-A_L-EN.XML	Title 2	EN	O
URN:S1000D:DMC-520-A	www.ex.com/ ➥DMC-520-A_L-FR.XML	Titre 2	FR	N
URN:S1000D:DMC-520-A_L-EN	www.ex.com/ ➥DMC-520-A_L-EN.XML	Title 2	EN	O
URN:S1000D:DMC-520-A_L-FR	www.ex.com/ ➥DMC-520-A_L-FR.XML	Titre 2	FR	O
URN:S1000D:ICN-00352	www.ex.com/ ➥ICN-00352.CGM	GraTitle 1	EN	O

Voici la requête appropriée pour être redirigé sur l'URL associée à `URN:S1000D:DMC-999-A` :

```
http://www.ex.com/resolution/resolveUrn?urn=URN:S1000D:DMC-999-A
```

Elle redirige vers l'URL

```
http://www.ex.com/DMC-999-A.XML.
```

Voici maintenant la requête pour renvoyer la caractéristique `title` de la ressource `URN:S1000D:DMC-999-A` :

```
http://www.ex.com/resolution/resolveUrn?urn=URN:S1000D:DMC-999-A&title
```

Elle renvoie le document XML suivant :

```
<rds>
<rdu scheme="dc">
  <rdf:rdf>
   <rdf:Description rdf:about="URN:S1000D:DMC-999-A">
     <urc name="titre">Title 1</urc>
   </rdf:description>
  </rdf:rdf>
</rdu>
</rds>
```

La requête qui porte sur `URN:S1000D:DMC-520-A`, sans précision alors qu'il existe plusieurs URL associées :

```
http://www.ex.com/resolution/resolveUrn?urn=URN:S1000D:DMC-520-A
```

redirigera vers la ressource par défaut :

```
http://www.ex.com/DMC-520-A_L-EN.XML
```

En revanche, si une spécification sur la métadonnée langue est fournie :

```
http://www.ex.com/resolution/resolveUrn?urn=URN:S1000D:DMC-520-
A&langue=FR
```

c'est l'URL de la ressource française qui sera choisie :

```
http://www.ex.com/DMC-520-A_L-FR.XML
```

Avec le mot-clé `multiple`, la requête retourne les métadonnées demandées de toutes les ressources associées à un URN. Dans l'exemple qui suit, la valeur de la méta-

donnée `title` est demandée pour toutes les ressources associées à `URN:S1000D:DMC-520-A`.

```
http://www.ex.com/resolution/resolveUrn?urn=URN:S1000D:DMC-520-
A&multiple&title
```

Cette requête retourne le document XML suivant :

```
<rds>
<rdu scheme="dc">
  <rdf:rdf>
   <rdf:Description rdf:about="URN:S1000D:DMC-520-A">
     <urc name="titre">Title 2</urc>
   </rdf:description>
  </rdf:rdf>
</rdu>
<rdu scheme="dc">
  <rdf:rdf>
   <rdf:Description rdf:about="URN:S1000D:DMC-520-A">
     <urc name="titre">Titre 2</urc>
   </rdf:description>
  </rdf:rdf>
</rdu>
</rds>
```

En utilisant `allurcs`, la requête suivante retourne les valeurs de toutes les métadonnées de la ressource associée à `URN:S1000D:DMC-999-A` :

```
http://www.ex.com/resolution/resolveUrn?urn=URN:S1000D:DMC-999-
A&allurcs
```

Cette requête renvoie le document XML suivant :

```
<rds>
<rdu scheme="dc">
  <rdf:rdf>
   <rdf:Description rdf:about="URN:S1000D:DMC-999-A">
    <urc name="url">http://www.ex.com/DMC-999-A.XML</urc>
    <urc name="titre">Title 1</urc>
    <urc name="langue">EN</urc>
   </rdf:description>
  </rdf:rdf>
</rdu>
</rds>
```

La requête qui retourne simplement l'URL d'une ressource utilise le paramètre `url` :

```
http://www.ex.com/resolution/resolveUrn?urn=URN:S1000D:ICN-00352&url
```

Cette requête renvoie le document XML suivant :

```
<rds>
<rdu scheme="dc">
  <rdf:rdf>
   <rdf:Description rdf:about="URN:S1000D:ICN-00352">
    <urc name="URL">http://www.ex.com/ICN-00352.CGM</urc>
   </rdf:description>
  </rdf:rdf>
</rdu>
</rds>
```

Pour résumer, les URN permettent de gérer les révisions uniquement si l'une ou l'autre des deux conditions suivantes est réunie :

1. L'indice de révision est codé dans l'identifiant de ressource cible.
2. L'indice de révision de la ressource cible est codé à l'extérieur du document, au niveau du serveur de résolution d'adresses URN.

Autres méthodes de gestion des révisions

Il existe au moins trois autres approches que nous ne ferons qu'évoquer ici :

- l'approche base de données XML,
- l'approche par calcul différentiel XUpdate,
- l'approche base de données relationnelle.

L'approche base de données XML

Cette approche consiste à charger le document dans une base de données en comparant nœud après nœud l'égalité de la nouvelle version avec l'ancienne.

En cas d'égalité, l'ancienne structure est conservée. En cas d'inégalité des nœuds, une nouvelle branche est créée dans la base.

Un arbre multidimensionnel, dit « multitemporel », se construit ainsi petit à petit. Il représente toutes les évolutions subies par les nœuds au fil du temps. N'importe quelle version du document peut être reconstruite à tout moment.

Dans cette approche, l'utilisateur qui produit un document XML ne s'occupe nullement de marquer ses révisions. Il se contente de modifier le document et de l'importer dans la base. C'est le système qui détecte seul les différences avec la version précédente, créant dans la base et de manière invisible pour l'utilisateur les marques correspondantes.

L'utilisateur peut à tout moment demander l'extraction de n'importe quelle version, ce qui ne signifie pas pour autant que les balises de révision seront introduites dans le document.

L'approche par calcul différentiel XUpdate

Cette approche est similaire à la précédente sauf que les documents XML révisés font l'objet d'une transformation importante au terme de laquelle le document est exprimé uniquement à l'aide d'un balisage servant à marquer les différences entre l'ancienne et la nouvelle version.

Pour cela, il « suffit » de calculer l'équivalent XUpdate des modifications effectuées dans le document originel. En effet, XUpdate est un langage XML qui permet de spécifier des ordres de modifications de documents XML.

Par exemple, le programme Xupdate suivant ajoute l'élément `address` après celui correspondant au chemin d'accès Xpath : `/addresses/address[1]` (repère ⓿). Ensuite, le programme ajoute un attribut `id` à l'élément nouvellement créé (repère ❶), puis ajoute enfin à l'identique les lignes repérées ❷.

```
<?xml version="1.0"?>
<xupdate:modifications version="1.0" xmlns:xupdate...>
 <xupdate:insert-after select="/addresses/address[1]"> ← ⓿
    <xupdate:element name="address">
      <xupdate:attribute name="id">2</xupdate:attribute> ← ❶
      <fullname>Lars Martin</fullname>       ← ❷
      <born day='2' month='12' year='1974'/> ← ❷
      <town>Leizig</town>                    ← ❷
      <country>Germany</country>             ← ❷
    </xupdate:element>
 </xupdate:insert-after>
</xupdate:modifications>
```

Puis, il ajoute un bloc de données entier au document XML suivant :

```
<?xml version="1.0"?>
<addresses version="1.0">
 <address id="1">
```

```
 <fullname>Andreas Laux</fullname>
 <born day='1' month='12' year='1978'/>
 <town>Leipzig</town>
 <country>Germany</country>
 </address>
</addresses>
```

qui devient :

```
<?xml version="1.0"?>
<addresses version="1.0">
 <address id="1">
    <fullname>Andreas Laux</fullname>
    <born day='1' month='12' year='1978'/>
    <town>Leipzig</town>
    <country>Germany</country>
 </address>
 <address id="2">
    <fullname>Lars Martin</fullname>
    <born day='2' month='12' year='1974'/>
    <town>Leizig</town>
    <country>Germany</country>
 </address>
</addresses>
```

L'inverse est vrai. Des produits permettent de stocker les révisions de documents XML sous la forme de programmes XUpdate : ils identifient les différences entre deux versions d'un même document XML, puis en déduisent le programme XUpdate correspondant. C'est alors le programme qui est conservé dans la base.

L'approche base de données relationnelle

Les bases de données relationnelles sont de mauvaises candidates au stockage des documents XML, en particulier, lorsque ces derniers contiennent des modèles mixtes, font appel à des espaces de noms variants et… quand il faut stocker plusieurs versions d'un même document XML dans la base.

La seule solution consiste à utiliser un modèle de table simple et à débiter le document XML en petits morceaux pour le faire tenir dans des champs élémentaires de la base de données.

Cette approche présente l'avantage de pouvoir stocker et identifier le moindre des caractères modifiés entre deux versions d'un même fichier XML, ce que ne peuvent faire les approches basées sur la logique XML. Certains caractères échappent com-

plètement à cette logique : il s'agit en particulier des blancs compris entre deux balises, au début ou à la fin du contenu d'un élément, des retours à la ligne, des caractères accentués qui peuvent être écrits tantôt sous leur forme Unicode (par exemple « é ») et tantôt sous forme d'entité (`é`).

Dans le cas où il faut vraiment identifier la moindre des modifications apportées à un document XML, la solution la plus appropriée est celle du tronçonnage ou débitage mécanique. Elle consiste à éclater le document en autant de petits bouts qu'il y a de balises XML, de contenus textuels et d'interstices entre les balises.

Par exemple, le tronçonnage du document précédent occasionnera les bouts présentés au tableau 13-2. Dans ce tableau, les numéros des nœuds correspondent à la méthode de calcul des identifiants de nœuds présentée au chapitre 4.

Tableau 13–2 Comparaison de deux documents XML par tronçonnage

Document d'origine				Document modifié			
N^0	**ss-N°**	**Type d'objet**	**Valeur**	**N^0**	**ss-N^0**	**Type d'objet**	**Valeur**
1,12	1	Élément	addresses	1,22	1	Élément	addresses
	2	Nom d'attribut	version		2	Nom d'attribut	version
	3	Valeur d'attribut	1.0		3	Valeur d'attribut	1.0
	4	Espace inter-unités	\n		4	Espace inter-unités	\n
2,11	1	Élément	address	2,11	1	Élément	address
	2	Nom d'attribut	Id		2	Nom d'attribut	Id
	3	Valeur d'attribut	1		3	Valeur d'attribut	1
	4	Espace inter-unités	\n		4	Espace inter-unités	\n
3,4	1	Élément	fullname	3,4	1	Élément	fullname
	2	Contenu textuel	Andreas Laux		2	Contenu textuel	Andreas Laux
	3	Espace inter-unités	\n		3	Espace inter-unités	\n
5,6	1	Élément	born	5,6	1	Élément	born
	2	Nom d'attribut	day		2	Nom d'attribut	day
	3	Valeur d'attribut	1		3	Valeur d'attribut	1
	4	Nom d'attribut	month		4	Nom d'attribut	month
	5	Valeur d'attribut	12		5	Valeur d'attribut	12
	6	Nom d'attribut	year		6	Nom d'attribut	year
	7	Valeur d'attribut	1978		7	Valeur d'attribut	1978
	8	Espace inter-unités	\n		8	Espace inter-unités	\n
7,8	1	Élément	town	7,8	1	Élément	town
	2	Contenu textuel	Leipzig				

Tableau 13–2 Comparaison de deux documents XML par tronçonnage

Document d'origine					Document modifié			
N°	ss-N°	Type d'objet	Valeur		N°	ss-N°	Type d'objet	Valeur
	3	Espace inter-unités	\n			2	Contenu textuel	Leipzig
9,10	1	Élément	country			3	Espace inter-unités	\n
	2	Contenu textuel	Germany		9,10	1	Élément	country
	3	Espace inter-unités	\n			2	Contenu textuel	Germany
						3	Espace inter-unités	\n
					12,11	1	Élément	address
						2	Nom d'attribut	Id
						3	Valeur d'attribut	2
						4	Espace inter-unités	\n
					13,14	1	Élément	fullname
						2	Contenu textuel	Lars Martin
					15,16	1	Élément	born
						2	Nom d'attribut	day
						3	Valeur d'attribut	2
						4	Nom d'attribut	month
						5	Valeur d'attribut	12
						6	Nom d'attribut	year
						7	Valeur d'attribut	1974
						8	Espace inter-unités	\n
					17,18	1	Élément	town
						2	Contenu textuel	Leipzig
						3	Espace inter-unités	\n
					19,20	1	Élément	country
						2	Contenu textuel	Germany
						3	Espace inter-unités	\n

Ainsi « aplatis », il est possible de comparer les deux tableaux et de se mettre en quête des différences. Un tel résultat peut facilement être transposé dans un modèle relationnel.

Cette approche a l'avantage de fonctionner simplement et efficacement. Le cas échéant, les documents seront préparés avant leur introduction dans la base. Le problème est que l'on a alors une liste de micro-différences qui ne permet pas de véritablement gérer les révisions.

En résumé

Dans ce chapitre, nous avons vu comment aborder le délicat problème de la gestion des révisions. Dans ce domaine, XML n'est pas forcément simple d'utilisation.

Principalement, il y a le choix entre :

- Marquer les révisions par un balisage spécifique à l'intérieur du document XML. La gestion se fait alors *a priori*.
- N'utiliser aucun marquage particulier et laisser des programmes calculer les différences entre les versions. La gestion des révisions s'effectue alors *a posteriori*.

Nous avons vu que la première approche pose un sérieux problème de conception des schémas et oblige à adopter des stratégies de gestion des révisions bien identifiées.

La seconde approche requiert que l'on dispose d'un outil ad hoc. Pour que les révisions puissent être mises en évidence lors de l'affichage ou la publication d'un document, il faudra alors que l'outil sache mettre dans le document le bon balisage (il ne suffit pas de savoir le calculer). Cette approche ne posera toutefois pas de problème de modélisation. Pour l'affichage ou la publication d'un document, on peut toujours se contenter d'une forme bien faite : il n'est pas nécessaire que cette forme soit valide.

Enfin, nous avons vu l'impact de la gestion des révisions sur les liens : là encore, des choix importants de conception devront être faits.

Par nature, les données de révisions et de versions ne se situent pas sur le même plan que les données et la structure des documents XML eux-mêmes. Une gestion limitée des révisions peut être établie à l'aide de balises respectant la séparation entre les données de versions et les données XML, comme l'a montré le modèle ATA. Cependant, une véritable gestion des révisions doit faire appel à des programmes ou des applications dédiées telles que les bases de données XML.

Représentation UML avancée pour XML Schema

Nous présentons dans cette annexe les notations UML à utiliser en regard de tous les mécanismes autorisés mais avancés de XML Schema. Nous analysons chaque difficulté et expliquons comment la traiter. Nous avons baptisé « avancés » les mécanismes de XML Schema rarement utilisés et dont on peut même se demander s'ils doivent raisonnablement être représentés dans les vues UML du système.

Voici les mécanismes avancés de XML Schema dont on trouvera ici la correspondance UML :

- les attributs et les types listes et unions,
- les attributs et autres valeurs par défaut,
- la définition d'annotations,
- les groupes d'attributs,
- les contraintes d'exclusion et les groupes de choix,
- les contraintes de simultanéité et les groupes simples,
- la question de l'ordre des éléments,
- la mixité des modèles.

Attributs et types listes de XML Schema

UML autorise à spécifier tant une multiplicité maximale qu'une multiplicité minimale pour les attributs UML. Les multiplicités servent à indiquer des contraintes sur le nombre de valeurs que doit avoir un attribut. La multiplicité minimale induit les contraintes suivantes :

- Lorsque la multiplicité minimale est égale à `0`, l'attribut est optionnel.
- Lorsque la multiplicité minimale est égale à `1`, l'attribut est obligatoire.
- Lorsque la multiplicité minimale est supérieure à `1`, l'attribut est une liste ayant un nombre de valeurs obligatoires égal à la valeur de la multiplicité minimale.

Par défaut, la multiplicité minimale est égale à `0`.

La multiplicité maximale induit les contraintes suivantes :

- Lorsque la multiplicité maximale est égale à `1`, l'attribut a une seule valeur.
- Lorsque la multiplicité maximale est supérieure à `1`, l'attribut est une liste de valeurs.

Par défaut, la multiplicité maximale est égale à `1`.

Pour savoir si un attribut a une seule valeur ou s'il est une liste, il suffit de regarder sa multiplicité maximale. La multiplicité minimale n'indique, quant à elle, que le nombre de valeurs obligatoires.

Les règles de représentation d'un attribut UML en XML dépendent du type de données de l'attribut et du choix de transformation des attributs en XML.

- Lorsqu'un attribut est de type composé, par exemple un type adresse avec rue, ville et code postal, l'attribut UML est obligatoirement représenté par un élément XML.
- Lorsqu'un attribut est de type simple, il peut au choix être représenté par un attribut XML ou par un élément XML. Le choix relève d'une décision de conception des schémas XML.

Lorsqu'un attribut UML est représenté par un élément XML, les valeurs des multiplicités minimale et maximale correspondent aux indicateurs d'occurrences de l'élément : `minOccur`, `maxOccur`.

Lorsqu'un attribut UML est représenté par un attribut XML, le type associé à l'attribut XML varie en fonction des valeurs des multiplicités. En effet, il n'est pas possible, en XML, de définir des indicateurs d'occurrences pour les attributs. En XML, un élément ne peut avoir qu'un seul attribut d'un nom donné. En revanche, cet attribut peut avoir une suite de valeurs. Il faut alors recourir au type liste de XML Schema. Le tableau A1-1 donne la liste des combinaisons possibles.

Tableau A–1 Représentation d'un attribut UML par un attribut XML

Exemple UML	Multiplicité minimale	Multiplicité maximale	Attribut XML	Exemple XML

partenaire 0..1
+telephone[0..1]:
telephoneType
	0	1	Attribut de type simple non obligatoire	`<xs:simpleType name="telephoneType">`
				`<xs:restrictionbase="xs:string"/>`
				`</xs:simpleType>`
				
				`<xs:element name="partenaire">`
				`  <xs:complexType>`
				`    <xs:attribute`
				`        name="telephone"`
				`        type="telephoneType"`
				`        use="`**`optional`**`"/>`
				`  </xs:complexType>`
				`</xs:element>`

partenaire 1..1
+telephone[1..1]:
telephoneType
	1	1	Attribut de type simple obligatoire	`<xs:simpleType name="telephoneType">`
				`<xs:restrictionbase="xs:string"/>`
				`</xs:simpleType>`
				
				`<xs:element name="partenaire">`
				`  <xs:complexType>`
				`    <xs:attribute`
				`        name="telephone"`
				`        type="telephoneType"`
				`        use="`**`required`**`"/>`
				`  </xs:complexType>`
				`</xs:element>`

partenaire 0..*
+telephone[0..*]:
telephoneType
	0	*	Attribut de type liste	`<xs:simpleType name="telephoneType">`
				`<xs:restrictionbase="xs:string"/>`
				`</xs:simpleType>`
				
				`<xsd:simpleType name="telListe">`
				`<xsd:list itemType="telephoneType"/>`
				`</xsd:simpleType>`
				
				`<xs:element name="partenaire">`
				`  <xs:complexType>`
				`    <xs:attribute`
				`        name="telephone"`
				`        type="`**`telListe`**`"`
				`        use="required"/>`
				`  </xs:complexType>`
				`</xs:element>`

Tableau A–1 Représentation d'un attribut UML par un attribut XML

Exemple UML	Multiplicité minimale	Multiplicité maximale	Attribut XML	Exemple XML
partenaire 0..4 +telephone[0..4]: telephoneType	0	>1 + Valeur limite	Attribut de type liste avec facette maxLength	<xs:simpleType name="telephoneType"> <xs:restrictionbase="xs:string"/> </xs:simpleType> <xs:simpleType name="telListe"> <xs:list itemType="telephoneType"/> </xs:simpleType> <xs:simpleType name="telListeMaxQuatre"> <xs:restriction base="telList"> **<xs:maxLength value="4"/>** </xs:restriction> </xs:simpleType> <xs:element name="partenaire"> <xs:complexType> <xs:attribute name="telephone" type="telListeMaxQuatre" use="required"/> </xs:complexType> </xs:element>
partenaire 3..4 +telephone[3..4]: telephoneType	>0 + Valeur limite	>1 + Valeur limite	Attribut de type liste avec facettes maxLength et min-Length	<xs:simpleType name="telephoneType"> <xs:restrictionbase="xs:string"/> </xs:simpleType> <xs:simpleType name="telListe"> <xs:list itemType="telephoneType"/> </xs:simpleType> <xs:simpleType name="telListeMinMax"> <xs:restriction base="telList"> <xs:minLength value="3"/> <xs:maxLength value="4"/> </xs:restriction> </xs:simpleType> <xs:element name="partenaire"> <xs:complexType>

Tableau A–1 Représentation d'un attribut UML par un attribut XML

Exemple UML	Multiplicité minimale	Multiplicité maximale	Attribut XML	Exemple XML

				`<xs:attribute`
				`name="telephone"`
				`type="telListeMinMax"`
				`use="required"/>`
				`</xs:complexType>`
				`</xs:element>`

À la lecture du tableau A-1, on aura pu noter que la correspondance UML/XML n'est pas directe. Ce qui est spécifié au niveau de l'attribut dans UML l'est au niveau du type de l'attribut dans XML.

En ce qui concerne le type union de XML Schema, on ne trouve pas de représentation correspondante dans UML. XML Schema dispose de capacités d'expression plus avancées en ce qui concerne le contrôle des valeurs d'attributs et d'éléments.

Attributs et valeur par défaut

Il est possible de donner une valeur par défaut aux attributs UML. La correspondance de ces valeurs par défaut en XML est donnée dans le tableau A-2.

La définition d'annotations

UML permet de définir des commentaires sur tout objet de modélisation. À chaque commentaire associé à un objet UML correspond dans le schéma XML une annotation XML de type documentation.

Groupe d'attributs

La notion de groupe d'attributs n'existe pas en UML. Le concept le plus proche est celui de type de données (*DataType* en anglais) avec la restriction suivante : les attributs du type de données doivent eux-mêmes être de type simple, c'est-à-dire ne pas se décomposer en d'autres attributs. Pour exprimer l'utilisation d'un groupe d'attributs par une classe, il faut utiliser une relation de généralisation entre la classe et le type de données.

Tableau A–2 Valeur d'attribut par défaut

Modèle UML	Représentation XML
Attribut avec valeur par défaut restaurant +menu:string=truite meunière	Cas où l'attribut UML donne lieu à un élément XML <xs:element name="restaurant"> <xs:complexType> <xs:sequence> <xs:element name="menu" type = "xs:string" default="truite meunière"/> </xs:sequence> </xs:c omplexType> </xs:element> Cas où l'attribut UML donne lieu à un attribut XML <xs:element name="partenaire"> <xs:complexType> <xs:attribute name="menu" type="xs:string" value="truire meunière"/> </xs:complexType> </xs:element>

Tableau A–3 Annotations

Modèle UML	Représentation XML
Une classe et ses commentaires Pilote Un premier commentaire Un deuxième commentaire	Annotation XML <xs:element name="pilote"> <xs:annotation> <xs:documentation>Un premier commentaire </xs:documentation> <xs:documentation>Un deuxième commentaire </xs:documentation> </xs:annotation> <xs:complexType> … </xs:complexType> </xs:element>

Sur le plan de la conformité à UML, cette relation de généralisation entre une classe et un type de données est inhabituelle, mais ne viole pas les règles de généralisation

établies par la norme UML 2.0. Si l'on avait considéré le groupe d'attributs comme une classe standard, nous aurions été confrontés à des difficultés pour la transposition du modèle de classe vers le modèle XML. Au chapitre 3, nous avons vu que, dans le cas général, chaque classe donne lieu à un élément XML ; or ce n'est pas ce que nous recherchons ici, les groupes d'attributs n'étant justement pas des éléments XML Schema.

Tableau A–4 Groupe d'attributs

Modèle UML

Une classe héritant d'un type de données

<<Data Type>>
ItemDelivery
+partNum:SKU
+shipBy:shipBy
+weightKg:Decimal

<<Enumeration>>
shipBy
+air
+any
+land

Item
+productName:string
+quantity:positiveInteger
+shipDate:Date
+USPrice:Decimal
+partNum:SKU
+shipBy:shipBy
+weightKg:Decimal

Représentation XML

Groupe d'attributs XML

```
<xsd:attributeGroup name="ItemDelivery">
  <xsd:attribute name="partNum" type="SKU"/>
  <xsd:attribute name="weightKg"
                 type="xsd:decimal"/>
  <xsd:attribute name="shipBy">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
      <xsd:enumeration value="air"/>
      <xsd:enumeration value="land"/>
      <xsd:enumeration value="any"/>
     </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:attributeGroup>
```

Tableau A–4 Groupe d'attributs

```
<xsd:element name="item">
<xsd:complexType>
  <xsd:sequence>
   <xsd:element name="productName"
                type="xsd:string"/>
   <xsd:element name="quantity">
    <xsd:simpleType>
     <xsd:restriction base="xsd:positiveInteger">
      <xsd:maxExclusive value="100"/>
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:element>
   <xsd:element name="USPrice"
                type="xsd:decimal"/>
   <xsd:element ref="comment"   minOccurs="0"/>
   <xsd:element name="shipDate" type="xsd:date"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="ItemDelivery"/>
</xsd:complexType>
</xsd:element>
```

La mixité des modèles

Tableau A–5 Modèle mixte

Modèle UML
Classe héritant du type de données string

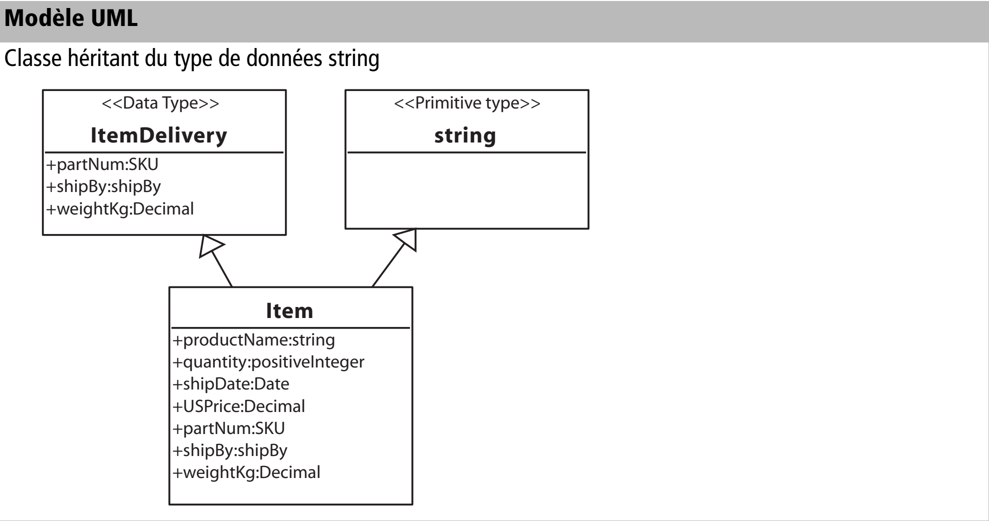


Tableau A–5 Modèle mixte

Représentation XML
Modèle de contenu mixte

```
<xsd:element name="item">
<xsd:complexType mixed=true>
  <xsd:sequence>
   <xsd:element name="productName"
                type="xsd:string"/>
   <xsd:element name="quantity">
    <xsd:simpleType>
     <xsd:restriction base="xsd:positiveInteger">
      <xsd:maxExclusive value="100"/>
     </xsd:restriction>
    </xsd:simpleType>
   </xsd:element>
   <xsd:element name="USPrice"
                type="xsd:decimal"/>
   <xsd:element ref="comment"   minOccurs="0"/>
   <xsd:element name="shipDate" type="xsd:date"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="ItemDelivery"/>
</xsd:complexType>
</xsd:element>
```

Dans un cas particulier d'héritage, le type de données hérité est le type primitif string. C'est un cas limite du point de vue de la conformité du modèle UML, même s'il reste valide. Il permet de représenter les modèles de contenu mixte de XML Schema comme indiqué dans le tableau A-5.

Comme pour d'autres correspondances entre modèles de classes et XML Schema, la solution proposée dans ce paragraphe pousse le modèle UML dans ses retranchements. Voilà une indication supplémentaire comme quoi il n'y a pas recouvrement un pour un du modèle de classes UML et de XML Schema.

Contrainte d'exclusion et groupe de choix

UML dispose d'une contrainte standard, appelée `{xor}`, permettant de définir une exclusion entre deux associations.

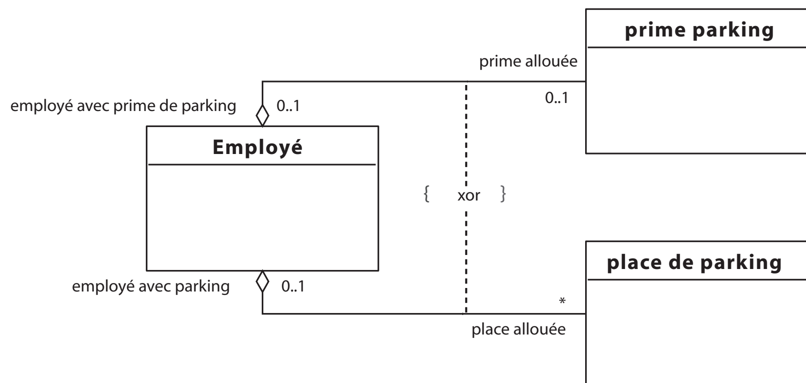
Les contraintes d'exclusion `xor` sont représentées par un groupe de choix XML.

> **VOCABULAIRE Contrainte d'exclusion**
>
> Une contrainte d'exclusion indique qu'un objet ne peut être relié que par l'une ou l'autre des associations référencées par la contrainte. Dans l'exemple du tableau A-6, un employé est soit un employé avec prime de parking, soit un employé avec place de parking ; ou, ce qui est équivalent, il a soit une prime de parking, soit une place de parking.

Tableau A–6 Contrainte d'exclusion et groupe de choix

Modèle UML

Contrainte d'exclusion



Représentation XML

Groupe de choix XML

```
<xs:element name="Employé">
  <xs:complexType>
    <xs:choice>
      <xs:element name="parkingAlloue"
                  type="placeDeParking"/>
      <xs:element name="primedeparkingAllouee"
                  type="primeDeParking"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Contraintes de simultanéité et groupes simples

Outre la contrainte standard `{xor}` de UML, il existe d'autres contraintes sur associations qui font l'objet d'un large consensus, en particulier la contrainte de simultanéité.

> **VOCABULAIRE Contrainte de simultanéité**
>
> Une contrainte de simultanéité indique que si un objet participe à l'une des associations de la contrainte, il participe alors également aux autres associations. Dans l'exemple suivant, si une commande est à facturer, elle est aussi à livrer, et réciproquement.

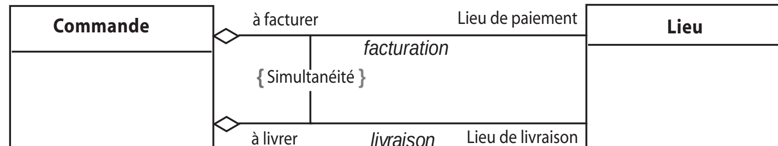


Figure A–1
Contrainte de simultanéité

Il est tentant d'utiliser directement les groupes simples XML (<xsd:group>) pour exprimer les contraintes de simultanéité dans les schémas XML. Malheureusement, contrairement aux groupes de choix, les groupes simples doivent être déclarés comme des éléments globaux pour être réutilisés dans plusieurs autres éléments XML. Tel n'est pas l'objectif de la contrainte de simultanéité qui vise seulement la présence simultanée d'associations et non pas leur réutilisation.

En UML, la réutilisation s'exprime toujours à l'aide de relations de généralisation. Il faut donc revoir le schéma de la figure A-1 en conséquence : une nouvelle classe abstraite indique la mise en facteur des associations facturation et livraison ; la contrainte de simultanéité est conservée. Une représentation XML possible de ce modèle de classe est présentée dans le tableau A-7.

Tableau A–7 Contrainte de simultanéité et groupe simple

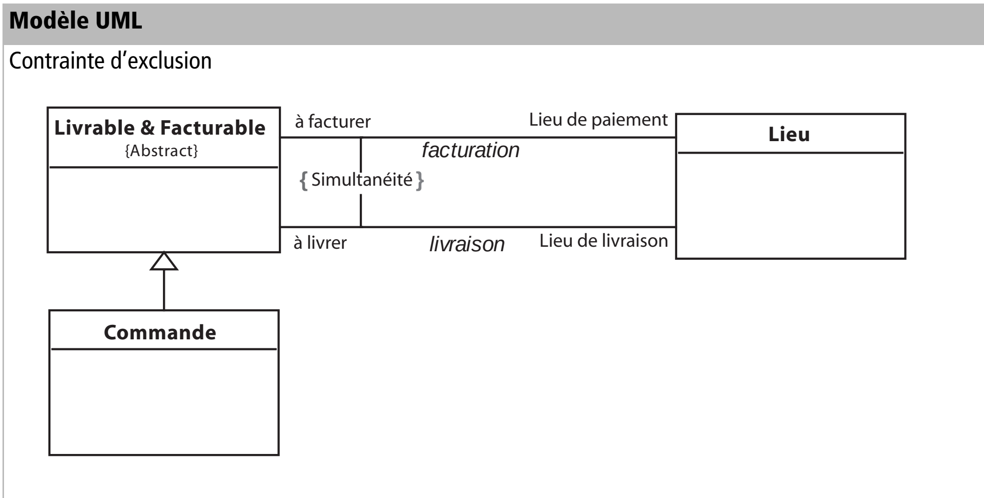
Modèle UML
Contrainte d'exclusion


Tableau A–7 Contrainte de simultanéité et groupe simple

Représentation XML

Groupe XML
 	
`<xs:schema elementFormDefault="qualified"`
`   attributeFormDefault="unqualified"`
`   xmlns:xl=http://www.w3.org/1999/xlink`
`   xmlns:xs="http://www.w3.org/2001/XMLSchema">`
`  <xs:import`
`     namespace=http://www.w3.org/1999/xlink`
`     schemaLocation="xlink.xsd"/>`
`  <xs:complexType name="entityRef">`
`    <xs:attribute ref="xl:type" fixed="simple"/>`
`    <xs:attribute ref="xl:href" use="required"/>`
`  </xs:complexType>`
`  <xs:group name="LivrableFacturable">`
`    <xs:sequence>`
`      <xs:element name="LieuLivraison"`
`                  type="entityRef"/>`
`      <xs:element name="LieuFacturation"`
`                  type="entityRef"/>`
`    </xs:sequence>`
`  </xs:group>`
`  <xs:element name="Commande">`
`    <xs:complexType>`
`      <xs:group ref="LivrableFacturable"/>`
`    </xs:complexType>`
`  </xs:element>`
`</xs:schema>`

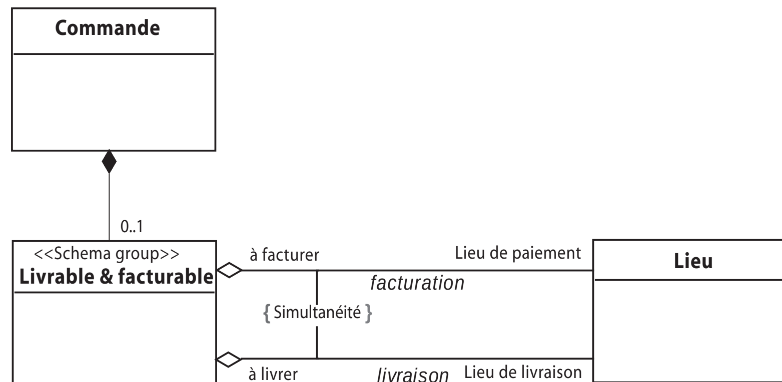
Il faut ici remarquer un nouveau cas où il n'y a pas de correspondance claire et directe entre le modèle de classes UML et XML Schema. Le tableau A-7 indique un choix particulier de transformation des relations de généralisation en XML, qui vient s'ajouter à celles exposées dans le chapitre 3.

Certaines recommandations conseillent l'utilisation de stéréotypes UML pour faire correspondre directement les concepts de XML Schema et ceux de UML.

> VOCABULAIRE **Stéréotypes UML**
>
> Les stéréotypes sont une technique de marquage des classes UML utilisée pour étendre le métamodèle UML et l'adapter à un cas d'emploi spécifique, par exemple XML Schema.

La figure A-2 montre l'exemple du stéréotype « Schema group » utilisé pour représenter un groupe XML à l'aide d'une classe UML spécifique. Cependant, l'usage de tels stéréotypes ne fait que singer les concepts XML dans les modèles de classes UML, sans apporter véritablement de valeur ajoutée. La classe `Livrable & Facturable` n'est pas à proprement parler une classe et ne fournit pas une aide à la découverte des classes entités et des modules de données, ce qui est l'objectif premier d'un modèle de conception UML, ainsi que nous l'avons exposé au chapitre 3. L'usage des stéréotypes pour représenter les spécificités de XML Schema tient plus de la recette de cuisine que de la véritable modélisation de données.

Figure A–2
Utilisation d'un stéréotype pour représenter un groupe XML



La question de l'ordre des éléments

Pour XML Schema, l'ordre des éléments revêt une grande importance et fait partie intégrante de la validation de la conformité des documents XML par rapport à leurs schémas. Tel n'est pas le cas des modèles de classes UML. Si l'on peut bien spécifier un ordre pour les attributs, les rôles d'associations et les généralisations d'une classe, cet ordre ne change pas substantiellement la nature du modèle de classe.

Les éléments XML correspondant à une classe UML proviennent soit d'attributs et de rôles d'associations directement reliés à cette classe, soit d'attributs et de rôles d'associations issus des classes héritées par les relations de généralisation. Il n'est donc pas possible d'obtenir un ordre absolu des éléments XML, mais seulement un ordre relatif : ordre des éléments provenant de généralisations + ordre des éléments provenant d'attributs + ordre des éléments provenant de rôles d'associations.

La règle le plus souvent appliquée est la suivante :

- Les éléments en provenance des classes héritées viennent en premier. Ils sont eux-mêmes ordonnés en fonction des deux règles suivantes.
- Les éléments en provenance d'attribut de classe viennent en second, dans l'ordre des attributs de la classe.
- Les éléments en provenance de rôle d'associations viennent en troisième, dans l'ordre des rôles de la classe.

Le tableau A-8 offre une illustration de l'application de ces règles.

Tableau A–8 Ordre des éléments

Modèle UML

classe sur-type
+attributS1:string
+attributS2:string
role S
roleT
*
*
classe associée T
classe
+attributC1:string
+attributC2:string
role1
propriétéX
*
0..1
classe aAssociée X
role2
propriétéA
*
*
classe associée A

Représentation XML

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified"
attributeFormDefault="unqualified" xmlns:xs="http://www.w3.org/2001/
XMLSchema">
  <xs:element name="Classe">
       <xs:complexType>
      <xs:sequence>
        <xs:element name="attributS1"/>
        <xs:element name="attributS2"/>
        <xs:element name="RoleT"/>
        <xs:element name="attributC1"/>
        <xs:element name="attributC2"/>
        <xs:element name="proprieteX"/>
        <xs:element name="proprieteA"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

B

Ressources

Textes normatifs

Au travers des chapitres de ce livre, un certain nombre de normes, qu'elles soient internationales du W3C ou sectorielles, ont été mentionnées, voire utilisées à titre d'exemple. Nous en donnons la liste au tableau suivant, accompagnée d'informations complémentaires, notamment des URL, qui permettront au lecteur d'approfondir ses connaissances.

Tableau B–1 URL des recommandations et normes mentionnées dans cet ouvrage

Norme ou recommandation	Éditeur	URL de la version utilisée pour ce livre et commentaires éventuels
S1000D v2.1	AECMA(3)	Spécification en date du 29 février 2004. http://www.s1000d.org/
ATA 2100	ATA(4)	Spécifications ATA pour les AMM (Aircraft Maintenance Manuals) http://www.airlines.org/public/publications Version 3.3 (décembre 1998) des spécifications ATA et version 3.0 de janvier 1999 de la DTD pour les AMM.

Tableau B–1 URL des recommandations et normes mentionnées dans cet ouvrage

Norme ou recommandation	Éditeur	URL de la version utilisée pour ce livre et commentaires éventuels
BPEL	OASIS(5)	Langage de spécification de processus exécutables proposé à l'origine par IBM et Microsoft sous le nom de BPEL4WS : Business Process Execution Language for Web Service. BPEL est un modèle décrivant un enchaînement de tâches, éventuellement réalisées à l'aide de services web. BPEL est depuis sa version 1.1 du 5 mai 2003 sous la responsabilité de l'organisation OASIS sous le nom de WSBPEL : Web Service Business Process Execution Language
BURNS	DSTC(16)	Basic URN Service résolution. http://archive.dstc.edu.au/rdu/TURNIP/burns.html
DocBook DTD V4.2CR3	OASIS(5)	http://www.oasis-open.org/docbook/ Copyright 1992-2002 HaL Computer Systems, Inc. O'Reilly & Associates, Inc., ArborText, Inc., Fujitsu Software Corporation, Norman Walsh, Sun Microsystems, Inc., and the Organization for the Advancement of Structured Information Standards (OASIS). $Id: docbook.dtd,v 1.14 2002/05/28 16:56:23 nwalsh Exp $ Permission to use, copy, modify and distribute the DocBook DTD and its accompanying documentation for any purpose and without fee is hereby granted in perpetuity, provided that the above copyright notice and this paragraph appear in all copies. The copyright holders make no representation about the suitability of the DTD for any purpose. It is provided "as is" without expressed or implied warranty.
DOM Requirements	W3C(1)	http://www.w3.org/TR/2000/WD-DOM-Requirements-20000412 Version de travail du 12 avril 2000 devenue une note de travail le 26 février 2004.
DSSSL	ISO(6)	Standard ISO 10179 : 1996, Document Style Semantics and Specification Language
FpML	ISDA(2)	Financial Products Markup Language version 1.0 : http://www.fpml.org/spec/2000/tr-fpml-arch-1-0-2000-09-25 Financial Products Markup Language version 2.0 : http://www.fpml.org/spec/2002/tr-fpml-2-0-2002-05-01 FpML est soumis à une licence publique d'utilisation, disponible à l'adresse http://www.fpml.org/documents/license Note technique sur la migration de XML à XML Schema : http://www.fpml.org/spec/2002/TBA

Tableau B–1 URL des recommandations et normes mentionnées dans cet ouvrage

Norme ou recommandation	Éditeur	URL de la version utilisée pour ce livre et commentaires éventuels
HyTime	ISO(6)	Standard ISO 10744 – Hypermedia Time based Structuring Language. Ch. Goldfarb, Eliot Kimber, Steve & Peter Newcomb
J2008	SAE(7)	http://www.xmlxperts.com/sae.htm
PURL	OCLC(14)	http:/purl.oclc.org/OCLC/PURL/SUMMARY
RDF	W3C(1)	RDF Vocabulary Description Language 1.0 : RDF Schema : http://www.w3.org/TR/2002/WD-rdf-schema-20020430/ Version de travail du 30 avril 2002 RDF Primer : http://www.w3.org/TR/2002/WD-rdf-primer-20020319/ Version de travail 19 mars 2002 RDF Schema Specification 1.0 : http://www.w3.org/2001/sw/RDFCore/Schema/20010913/ Version de travail, September @@XX 2001 (date exacte non précisée) RDF/XML Syntax Specification (Revised) : http://www.w3.org/TR/2002/WD-rdf-syntax-grammar-20020325 Version de travail du 25 mars 2002 RDF Model and Syntax Specification : http://www.w3.org/TR/1999/REC-rdf-syntax-19990222 Recommandation du 22 février 1999 RDF Schema Specification 1.0 : http://www.w3.org/TR/2000/CR-rdf-schema-20000327 Candidate Recommandation du 27 mars 2000
Relax	INSTAC(8)	MURATA Makoto — Regular Language description for XML — 2001 http://www.xml.gr.jp/relax/

Tableau B–1 URL des recommandations et normes mentionnées dans cet ouvrage

Norme ou recommandation	Éditeur	URL de la version utilisée pour ce livre et commentaires éventuels
Relax NG	OASIS(5)	Version du 3 décembre 2001 http://www.oasis-open.org/committees/relax-ng/spec-20011203.html Éditée par : James Clark et MURATA Makoto Copyright © The Organization for the Advancement of Structured Information Standards [OASIS] 2001. All Rights Reserved. This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to OASIS, except as needed for the purpose of developing OASIS specifications, in which case the procedures for copyrights defined in the OASIS Intellectual Property Rights document must be followed, or as required to translate it into languages other than English.
RFC 822	IETF(9)	Standard de l'IETF de 1982 définissant le format des fichiers de courrier électronique. http://www.ietf.org/rfc.html
Schematron	ASCC(10)	« The Schematron Assertion Language 1.5 » Spécification du 1er octobre 2002. Sites à consulter : http://www.ascc.net/xml/resource/schematron/schematron.html http://sourceforge.net/projects/schematron
SGML	ISO(6)	ISO 879-986 (édition) et ISO 8879:1979 (édition).
SOAP	W3C(1)	Recommandation depuis le 24 juin 2003. Version 1.1 : http://www.w3.org/TR/2000/NOTE-SOAP-20000508/ Version 1.2 Part 0: Primer : http://www.w3.org/TR/2003/REC-soap12-part0-20030624/ Version 1.2 Part 1: Messaging Framework : http://www.w3.org/TR/2003/REC-soap12-part1-20030624/ Version 1.2 Part 2: Adjuncts : http://www.w3.org/TR/2003/REC-soap12-part2-20030624/

Tableau B–1 URL des recommandations et normes mentionnées dans cet ouvrage

Norme ou recommandation	Éditeur	URL de la version utilisée pour ce livre et commentaires éventuels
Topics Map	ISO(6)	ISO 13250:1998 Michel Biezunski, Martin Bryan et Steve Newcomb
TREX	TOSSC(11)	2001 James Clark Tree Regular Expressions for XML http://www.thaiopensource.com/trex/
UBL	OASIS(5)	Proposition de OASIS du 17 novembre 2003 visant à fournir un modèle d'échange des documents relatifs aux transactions commerciales (commandes, facturations…) http://www.oasis-open.org/committees/ubl/lcsc/UBLv1-beta
UDDI version 2.0	UDDI.org (12)	Standard de UDDI.org initialement proposé par Ariba, IBM, Microsoft (septembre 2000). La version Open Draft Specification 2.0 de juin 2001 a été réalisée par IBM, Microsoft et SAP. http://www.uddi.org/pubs/ DataStructure-V2.00-Open-20010608.pdf Copyright © 2001 by Accenture, Ariba, Inc., Commerce One, Inc., Compaq Computer Corporation, Equifax, Inc., Fujitsu Limited, Hewlett-Packard Company, i2 Technologies, Inc., Intel Corporation, International Business Machines Corporation, Microsoft Corporation, Oracle Corporation, SAP AG, Sun Microsystems, Inc., and VeriSign, Inc. All Rights Reserved http://www.uddi.org/pubs/ DataStructure-V2.00-Open-20010608.pdf version 3 : http://uddi.org/pubs/uddi_v3.htm
Infrastructure UML 2.0	OMG(13)	Infrastructure de UML 2.0 Le statut de la spécification est : « Final Adopted Specification ». Un dernier vote du processus de publication de l'OMG devrait donner son statut officiel à la spécification, totalement publique avant la fin de l'année 2005. http://www.omg.org/cgi-bin/doc?ptc/2003-09-15
Superstructure UML 2.0	OMG(13)	Superstructure de UML 2.0 Le statut de la spécification est : « Final Adopted Specification ». Un dernier vote du processus de publication de l'OMG devrait donner son statut officiel à la spécification, totalement publique avant la fin de l'année 2005. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02

Tableau B–1 URL des recommandations et normes mentionnées dans cet ouvrage

Norme ou recommandation	Éditeur	URL de la version utilisée pour ce livre et commentaires éventuels
MOF 2.0	OMG(13)	Meta Object facility version 2.0 Le statut de la spécification est : « Final Adopted Specification ». Un dernier vote du processus de publication de l'OMG devrait donner son statut officiel à la spécification, totalement publique avant la fin de l'année 2005. http://www.omg.org/cgi-bin/doc?ptc/2003-10-04
URI	IETF(9)	RFC 2396 d'août 1998. http://www.ietf.org/rfc.html
URL	IETF(9)	RFC 1738 de décembre 1994. http://www.ietf.org/rfc.html
URN	IETF(9)	RFC 2141 de mai 1997. http://www.ietf.org/rfc.html
UUID	ISO(6)	ISO/IEC 11578:1996 Calcul des identifiants uniques universels
WebDAV	IETF(9)	RFC 2518 de février 1999 et 3253 de mars 2002. http://www.ietf.org/rfc.html
WSDL	W3C(1)	La spécification 1.0 du 25 septembre 2000 a donné lieu à une note du W3C 1.1 le 15 mars 2001, éditée par Microsoft et IBM. http://www.w3.org/TR/2001/NOTE-wsdl-20010315 Les textes sont actuellement à l'état de dernière version de travail depuis le 3 août 2004. http://www.w3.org/TR/2004/WD-wsdl20-20040803/ http://www.w3.org/TR/2004/WD-wsdl20-extensions-20040803/ http://www.w3.org/TR/2004/WD-wsdl20-bindings-20040803/
XMI 2.0	OMG(13)	XML Metadata Interchange 2.0 Format d'échange des modèles MOF-UML http://www.omg.org/cgi-bin/doc?formal/2003-05-02
XML Canonical	W3C(1)	http://www.w3.org/TR/2001/REC-xml-c14n-20010315 Recommandation du 15 mars 2001
XML	W3C(1)	http://www.w3.org/TR/2000/WD-xml-2e-20000814 Version de travail du 14 août 2000
XML Infoset	W3C(1)	http://www.w3.org/TR/2001/REC-xml-infoset-20011024 Recommandation du 24 octobre 2001

Tableau B–1 URL des recommandations et normes mentionnées dans cet ouvrage

Norme ou recommandation	Éditeur	URL de la version utilisée pour ce livre et commentaires éventuels
XML Namespace	W3C(1)	Recommandation du 14 janvier 1999 http://www.w3.org/TR/1999/REC-xml-names-19990114
XML Schema	W3C(1)	Vous trouverez les textes de la recommandation XML Schema aux adresses http://www.w3.org/TR/2001/REC-XMLschema-0-20010502/ http://www.w3.org/TR/2001/REC-XMLschema-1-20010502/ http://www.w3.org/TR/2001/REC-XMLschema-2-20010502/ pour les versions anglaises des tomes 0, I et II, et aux adresses http://XMLfr.org/w3c/TR/XMLschema-0/ http://XMLfr.org/w3c/TR/XMLschema-1 pour les versions françaises des tomes 0 et I.
XPath 1.0	W3C(1)	Recommandation du W3C du 16 novembre 1999 Version française : http://xmlfr.org/w3c/TR/xpath/ Version anglaise : http://www.w3.org/TR/1999/REC-xpath-19991116
XPath 2.0	W3C(1)	Version de travail du 30 avril 2002 http://www.w3.org/TR/2002/WD-xpath20-20020430
XSL v1.1	W3C(1)	Document de travail du 17 décembre 2003. http://www.w3.org/TR/2003/WD-xsl11-20031217/
XSL-fo	W3C(1)	Recommandation du 15 octobre 2001 définissant un vocabulaire XML pour le formatage des documents XML.
XSLT 1.0	W3C(1)	Recommandation du 16 novembre 1999 définissant un langage XML pour programmer des opérations de transformation de documents XML. XSLT est très utilisé pour transformer des données XML en données HTML affichées dans un navigateur, mais cela n'est pas la seule application possible. Version française : http://xmlfr.org/w3c/TR/xslt/ Version anglaise : http://www.w3.org/TR/1999/REC-xslt-19991116
XSLT v2.0	W3C(1)	Version de travail du 12 novembre 2003 http://www.w3.org/TR/xslt20/
XQuery	W3C(1)	XQuery n'est pas encore une recommandation (le dernier document de travail est daté du 4 avril 2005). http://www.w3.org/TR/2005/WD-xquery-20050404/ XQuery est une transposition en XML du langage SQL qui vise à interroger des bases de données XML de la même manière que le SQL est utilisé pour interroger les bases de données relationnelles.

Tableau B–1 URL des recommandations et normes mentionnées dans cet ouvrage

Norme ou recommandation	Éditeur	URL de la version utilisée pour ce livre et commentaires éventuels
XUpdate	Xmldb.org	L'objet de Xupdate est de spécifier en XML un langage de mise à jour des données stockées dans des bases de données XML. Le texte est à l'état de document de travail depuis le 14 septembre 2000. http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html Le W3C a annoncé que XUpdate serait une base de travail pour mettre au point un langage de mise à jour de documents XML mais qu'il ne pouvait être intégré à XQuery en l'état (Jonathan Robie, 30 mai 2002).

1 World Wide Web Consortium – Copyright© 1999 W3C (MIT, INRIA, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply
2 International Swaps and Derivatives Association, Inc.
3 Association européenne des constructeurs de matériel aéronautique
4 Air Transport Association
5 Organization for the Advancement of Structured Information Standards
6 International Standard Organization
7 Society for Automotive Engineers
8 Information Technology Research and Standardization Center
9 Internet Engineering Task Force
10 Academia Sinica Computing Centre
11 Thai Open Source Software Center
12 Universal Description Discovery & Integration Organization
13 Object Management Group
14 Online Library Computer Center
15 Business Process Management Initiative
16 Cooperative Research Centre for Distributed Systems Technology

Bibliographie

Aux textes normatifs susmentionnés, il convient d'ajouter un certain nombre d'ouvrages et articles.

A schema for serialized infosets, R. Tobin, H. Thompson, LTG, University of Edinburgh, http://www.w3.org/2001/05/serialized-infoset-schema.html

UML, M. Fowler, CampusPress, ISBN 2-7440-1090-1, avril 2001

Modélisation d'applications XML avec UML, D. Carlson, Eyrolles, ISBN 2-212-09297-0, octobre 2001

State of California, Air Resources Board, Final statement of reasons for rulemaking including summary of comments and agency response, considered : december 12, 1996 Agenda Item No : 96-10-2

An XML version of SAE J2008, Version 1.01 XML, D. Kennedy, 3/3/99, http://www.xmlxperts.com/saexml.htm

Storing XML in relational databases, I. Dayen, 20 juin 2001, http://www.xml.com/pub/a/2001/06/20/databases.html

XML and databases, R. Bourret, février 2002, http://www.rpbourret.com/xml/ XMLAndDatabases.htm

XML database products, R. Bourret, 14 mai 2002, http://www.rpbourret.com/xml/ProdsCMS.htm

Mapping DTDs to databases, R. Bourret, 2000, http://www.xml.com/lpt/a/2001/05/09/dtdtodbs.html

Content model algebra, Exoterica Corporation, http://home.chello.no/~mgrsby/sgmlintr/file0003.htm

XML Schema, E. van der Vlist, O'Reilly, ISBN 2-84177-215-2

XML, A. Michard, Eyrolles, ISBN 2-212-09052-8

Services Web avec SOAP, WSDL, UDDI, ebXML, J.-M. Chauvet, Eyrolles, ISBN 2-212-11047-2

XML et les bases de données, K. Williams, M. Brundage, P. Dengler, J. Gabriel, A. Hoskinson, M. Kay, T. Maxwell, M. Ochoa, J. Papa, M. Vanmane, Eyrolles, ISBN 2-212-009282-2

La théorie du chaos – Vers une nouvelle science, J. Gleick, traduction de C. Jeanmougin, Flammarion, ISBN 2-08-081219-X

Introducing the Schematron, Uche Ogbuji, ITWorld, http://itworld.com

Bases de données : des systèmes relationnels aux systèmes à objets, Cl. Delobel, Ch. Lécluse, Ph. Richard, InterEditions, sept. 1991 ISBN 2-7296-0371-9

Schémas XML, J.-J. Thomasson, Eyrolles, ISBN 2-212-11195-9

Services Web avec J2EE et .NET, L. Maesano, Ch. Bernard, X. Le Galles, Eyrolles 2003, ISBN 2-212-11067-7

Sigles et acronymes

Sigles et acronymes	Forme développée
AECMA	Association Européenne des Constructeurs de Matériel Aéronautique
ASD	AeroSpace and Defence Industries Association of Europe
ATA	Air Transport Association
BPEL	Business Process Execution Language.
BPEL4WS	Business Process Execution Language for Web Services.
BPMI	Business Process Management Initiative
BPMN	Business Process Modeling Notation
BPQL	Business Process Query Language
BURNS	Basic URN Service.
DSTC	Cooperative Research Centre for Distributed Systems Technology
FPML	Financial Product Markup Language
IETF	Internet Engineering Task Force
ISDA	International Swaps and Derivatives Association
MOF	Meta Object Facility
NAICS	North American Industry Classification System
OCLC	Online Computer Library Center
OASIS	Organization for the Advancement of Structured Information Standards

Sigles et acronymes	Forme développée
OMG	Object Management Group
PURL	Permanent URL
RDF	Resource Description Framework
RDU	Resource Description Unit
SGML	Standard Generalized Markup Language
SOAP	Simple Object Access Protocol
TREX	Tree Regular Expressions for XML
UDDI	Universal Description Discovery & Integration
UBL	Universal Business Language.
UML	Unified Modeling Language
UNSPSC	Universal Standard Products and Services Classification
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Names
URC	Uniform Resource Characteristic
UUID	Universally Unique ID
W3C	World Wide Web Consortium
WAP	Web Access Protocol
WebDAV	Distributed Authoring and Versioning Protocol for the World Wide Web
WML	Wireless Markup Language.
WSBPEL	Web Service Business Process Execution Language
WSDL	Web Services Description Language
XMI	XML Metadata Interchange
XML	EXtended Markup Language
XQuery	XML Query language
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations
XTM	XML Topics Map
XUpdate	XML Update language

D

Infoset

Vous trouverez ici une description détaillée du modèle Infoset du W3C. Elle vient en complément de l'introduction faite à ce modèle dans le chapitre 2.

Unité d'information de type document

L'unité d'information document est celle par laquelle commence le document XML. Souvent, elle est appelée prologue.

Cette unité d'information a les propriétés suivantes :

- Des enfants pouvant être: une unité d'information de type élément qui correspond à la racine du document XML, une instruction de traitement par instruction de traitement se trouvant en dehors du contenu même du document XML et de la DTD éventuellement présente en début de document XML, une unité d'information de type déclaration de type de document quand une telle déclaration est présente dans le prologue du document XML.
- Le cas échéant une propriété notations : constituée exactement d'une unité d'information de type notation par notation déclarée dans la DTD.
- Des entités non analysées : exactement une par entité non analysée déclarée dans la DTD.
- Un URI de base : celui de l'entité document.

- Un type de codage des caractères : celui déclaré dans l'entité document (concrètement, le type de codage des caractères est déclaré *via* la déclaration XML sous la forme `<?xml version="1.0" encoding="utf-8"?>` par exemple).
- Un indicateur de complétude dont la valeur peut être `yes` ou `no`. Il indique si le document XML a déjà été analysé et canonisé ou si des valeurs par défaut restent à découvrir au travers de la DTD ou du schéma.
- Un attribut de version représentant la version de XML utilisée dans le document.
- Un indicateur de traitement des déclarations d'entités qui ne fait pas partie vraiment du document mais que le parseur renseigne pour indiquer s'il a lu ou non la DTD.

Unité d'information de type élément

Les unités d'information de type élément correspondent aux balises contenues dans un document XML. En particulier, l'élément racine est une unité d'information de type élément qui fait l'objet d'une propriété de l'unité d'information de type document.

Cette unité d'information a les propriétés suivantes :

- Un espace de noms qui est renseigné quand l'élément est rattaché à un espace de noms spécifique.
- Un nom local qui est le nom de l'élément diminué de l'éventuel préfixe représentant un espace de noms et du signe « deux-points » qui va avec.
- Le cas échant un préfixe qui est celui associé à l'espace de noms.
- Un ensemble d'enfants qui représentent chacun l'une des unités d'information constitutives du corps du document XML. Ensemble ordonné selon leur ordre d'apparition dans le document.
- Un ensemble d'unités d'information de type attribut, exactement une par attribut porté explicitement ou implicitement par l'élément.
- Un ensemble d'unités d'information de type attributs représentant des déclarations d'espaces de noms : exactement une par déclaration d'espace de noms, y compris si un espace de noms par défaut est déclaré.
- Une liste d'espaces de noms valides pour l'élément courant.
- L'URI de base valable pour l'élément.
- Le parent de l'élément courant.

Unité d'information de type attribut

Les unités d'information de type attribut correspondent aux attributs portés par les éléments. Elles sont utilisées pour tous les attributs, y compris ceux définis avec une valeur par défaut dans la DTD ainsi que l'attribut spécial de déclaration d'espace de noms.

Cette unité d'information a les propriétés suivantes :

- Le cas échant un nom d'espace de noms.
- Un nom local.
- Un préfixe.
- Une valeur normalisée : il s'agit de la valeur de l'attribut suite à sa mise en forme lexicale conformément aux règles de normalisation édictées par XML ou les catalogues de types de XML Schema.
- Un indicateur permettant de savoir si l'attribut était présent dès le départ dans le document XML où s'il a été ajouté par le parseur (ce qui arrive quand l'attribut a une valeur par défaut attribuée par la DTD ou le schéma XML).
- Un indicateur du type de l'attribut.
- Un indicateur de référence pour les attributs qui sont de type identificateur ou référence à une entité ou une notation.
- Un indicateur permettant de connaître l'élément propriétaire de l'attribut.

Unité d'information de type instruction de traitement

Ce type d'unité d'information correspond aux instructions de traitement présentes dans le document XML.

Il a les propriétés suivantes :

- Une cible.
- Un contenu.
- Un URI de base.
- Le cas échéant une notation correspondant au nom de la cible.
- Un parent qui est l'unité d'information élément de type document ou définition de type de document.

Unité d'information de type référence d'entité non résolue

L'unité d'information de type référence d'entité non résolue est utilisée quand le processeur XML n'a pas expansé une entité externe.

Cette unité d'information a les propriétés suivantes :

- Le nom de l'entité référencée.
- Le cas échéant l'identifiant système de l'entité.
- Le cas échéant l'identifiant public de l'entité.
- Le cas échéant l'URI de base de l'identifiant public.
- L'unité d'information parent qui la contient.

Unité d'information de type caractère

Il existe une unité d'information de type caractère par caractère de donnée du document, même ceux qui apparaissent sous la forme de référence de caractère.

Cette unité d'information a les propriétés suivantes :

- Le code ISO 10646 du caractère.
- Un booléen qui indique si un espace blanc valide fait partie du contenu du document ou non.
- Un parent qui est une unité d'information de type élément.

Unité d'information de type commentaire

Il existe une unité d'information de type commentaire par commentaire présent dans le document, à l'exception de ceux se trouvant dans la DTD du document.

Cette unité d'information a les propriétés suivantes :

- Un contenu constitué de la chaîne de caractères formant le commentaire à proprement parler.
- Un parent qui est une unité d'information de type élément ou document.

Unité d'information de type déclaration de document

L'unité d'information de type déclaration de document existe quand le document XML a une déclaration de type de document. Les éventuelles déclarations d'entités et notations définies dans la déclaration de type de document sont des propriétés de l'unité d'information de type document et non de celle que nous sommes en train d'écrire.

Cette unité d'information a les propriétés suivantes :

- Le cas échéant un identificateur système permettant de référencer l'éventuel corps de la DTD qui peut être stocké dans un fichier à part.
- Le cas échéant un identificateur public.
- Des enfants composés des unités d'information de type instruction de traitement présentes dans la DTD.
- L'unité d'information parent de type document.

Unité d'information de type entité non analysée

Une unité d'information de type entité non analysée existe pour chaque entité générale déclarée dans la DTD.

Cette unité d'information a les propriétés suivantes :

- Le nom de l'entité.
- Le cas échéant un identificateur système.
- Le cas échéant un identificateur public.
- Le cas échéant l'URI de base de l'identificateur système.
- Le cas échéant un nom de notation.

Unité d'information de type notation

Une unité d'information de type notation existe pour chaque notation définie dans la DTD.

Cette unité d'information a les propriétés suivantes :

- Un nom.
- Le cas échéant un identificateur système.
- Le cas échéant un identificateur public.
- Le cas échéant l'URI de base de l'identificateur système.

Unité d'information de type espace de noms

Chaque élément du document a comme propriété une unité d'information de type espace de noms pour chaque espace de noms en vigueur au niveau de l'élément.

Cette unité d'information a les propriétés suivantes :

- Le préfixe associé à l'espace de noms.
- Le nom de l'espace de noms que ce préfixe représente.

Glossaire

Accessoire polymorphe

Cette locution est relative au modèle SOAP utilisé dans les services web. Elle désigne une manière de transmettre, sous forme XML, des tableaux à n dimensions contenant un nombre variable de types de données. Ce mécanisme original n'est supporté naturellement ni par XML ni par XML Schema.

Adressage

On désigne par adressage tout ce qui concourt à connaître et exploiter le lieu de stockage d'une donnée ou d'une ressource. Le plus souvent, il s'agit d'un chemin d'accès, mais cela peut prendre des formes plus sophistiquées en XML, par exemple avec Xpointer ; technique qui permet de combiner un chemin d'accès dans un système de fichiers avec un chemin d'accès à l'intérieur d'un document XML.

Agrégation

Terme issu du monde UML, une agrégation est un type d'association entre deux classes exprimant un rapport entre un tout et ses parties. La classe formant le tout est appelée « agrégat », l'autre classe désigne les parties. Les agrégations indiquent ainsi un rapport structurel entre les classes.

Par exemple, on considérera une famille comme un tout et les membres de la famille comme les parties.

Voir composition.

Ambiguïté

L'ambiguïté est l'incapacité à dire si un document XML est valide ou non. Le plus souvent, les validateurs, ou parseurs, pourraient lever l'ambiguïté en allant voir plus en avant ce que le document XML contient ; cela s'appelle le *lookahead*, mais c'est interdit par l'ensemble des normes et recommandations SGML et XML.

Les expressions « modèles ambigus » de SGML, « modèle de contenu non déterministe » de XML 1.0 et « règle de la particule unique » désignent le même problème d'ambiguïté dont XML Schema donne la définition suivante :

« *Un modèle de contenu doit être formé de telle manière que, pendant la validation·d'une séquence d'unités d'information de type élément, la particule, contenue directement, indirectement ou implicitement, avec laquelle on essaie de·valider·l'une après l'autre chaque unité de la séquence, doit pouvoir être identifiée sans avoir à examiner le contenu ou les attributs de chaque unité et sans avoir besoin de faire appel aux informations relatives aux autres unités du reste de la séquence.* »

Appel d'entité

Cette locution appartient aux mondes SGML et XML. Un appel d'entité désigne l'utilisation d'un nom d'entité dans un document XML. Par exemple, soit une entité à laquelle on donne le nom `texte`, alors l'appel d'entité correspondant sera la chaîne de caractère `&texte;`.

```
<!DOCTYPE sample [
<!ENTITY texte "Ceci est un paragraphe utilisant un appel d'entité.">
............
]>
<root>
<p>&texte;</p>
</root>
```

Applicabilité

L'applicabilité est l'identification de l'objet auquel s'applique une information. La gestion de l'applicabilité peut prendre des formes simples, tel un simple nom de système, ou complexes lorsque ledit système possède des variantes techniques (« applicable pour les outils de la marque untel… »), temporelles (« applicable du tant ou tant… »), calendaires (« applicable à partir de telle date… ») et géographiques (« applicable aux systèmes produits en Espagne… »). Ces critères d'applicabilité peuvent être combinés entre eux.

Arbre, arborescence

Le concept de classification de l'information sous la forme d'arbre n'est pas nouveau. Tous les systèmes mécaniques ou logiques sont tôt ou tard, dans leurs phases de conception, décomposés en objets élémentaires prenant la forme d'une arborescence.

En XML, l'organisation arborescente de l'information est évidente, mais dans les représentations UML des modèles de données, une telle représentation arborescente n'est pas un postulat de base. En effet, une représentation arborescente n'est suffisante ni pour représenter la manière dont les données interagissent les unes avec les autres, ni les liens entre ces mêmes données.

Architecture

Par analogie avec l'art de construire des édifices, l'architecture en informatique désigne les principes et techniques d'organisation des systèmes informatiques ; par exemple, l'architecture en trois tiers, l'architecture de service, etc.

Articulation

Par analogie avec le corps humain, l'articulation est un mécanisme qui donne de la souplesse à un modèle :

- Le modèle peut intégrer facilement d'autres modèles.
- Le modèle peut valider des documents XML de plus petite taille que le modèle lui-même.

Attributs flottants

Un attribut est dit flottant quand on peut le mettre n'importe où dans la structure sans déclaration préalable. Actuellement, XML Schema permet d'y recourir seulement pour les attributs de l'espace de noms XML Schema Instance, reconnu par le préfixe `xsi` qui lui est souvent associé. Les trois attributs les plus connus de XML Schema Instance sont `xsi:type`, `xsi:noNamespaceSchemaLocation` et `xsi:schemaLocation`. Un autre attribut flottant connu est `xmlns` qui définit un préfixe d'espace de noms.

Cela devrait également devenir le cas à terme avec les attributs de XLink.

Backbone

Encore appelé structure d'assemblage ou squelette, un backbone est un document XML dont la seule vocation est de définir un assemblage (ordonné ou non) de documents XML. Il s'agit donc essentiellement d'un document XML qui contient des pointeurs vers d'autres documents XML.

Base de données

Le nom complet en est *Système de gestion de base de données*, ou SGBD. Expression trop longue à dire et écrire, l'habitude a été prise d'utiliser la forme raccourcie *base de données* ; en revanche, le sigle SGBD est, quant à lui, resté.

Tenant compte de ce contexte, une base de données est un logiciel capable de gérer un volume, en général important, de données. Gérer signifie ici : contrôler les imports et exports de données, l'accès aux données, leur mise à jour, leur cohérence, et offrir des outils de calcul capables de s'appuyer sur la structure logique des données pour réaliser des opérations (calculs arithmétiques, booléens, recherche par motifs lexicaux ou valeurs…).

Base de données XML

Il s'agit d'une base de données dédiée au modèle XML. Cela signifie que les données sont stockées dans des documents XML, que les fonctions d'extraction et de mise à jour sont celles définies pour les données XML (XQuery, Xupdate, DOM, XSLT, SAX), que les méthodes d'accès sont compatibles avec un protocole tel que SOAP et que les standards définis par le W3C ou l'ISO sont tout ou partie mis en œuvre.

Il existe :

- des bases « natives XML » développées sur le seul modèle XML ;
- des bases relationnelles dans lesquelles le XML est projeté (on dit « mappé »), ces bases étant restrictives car les modèles XML et relationnels ne sont pas égaux ;
- des bases XML qui utilisent un moteur objet ;
- des bases XML qui s'appuient sur des bases hiérarchiques.

Cardinalité

On appelle cardinalité le nombre d'éléments d'un ensemble. En UML et XML, cela représente le nombre de fois qu'un objet peut être utilisé relativement à un autre.

Carte ou cartographie de sujets, de topiques

Expressions utilisées pour traduire les termes Topics Map. Elles désignent un modèle XML bien particulier dont la vocation est de définir les relations qui unissent des ressources informatiques, physiques ou virtuelles, ainsi que les descripteurs de ces dernières.

Cible

Une cible est la ressource informatique, physique ou logique pointée par un lien, une relation ou tout mécanisme de référencement ou de liaison.

Classe

Une classe est un ensemble concret ou logique qui contient tous les objets d'un même type, voire d'un type dérivé ; on parle alors de classes et de sous-classes.

La classe est concrète lorsqu'elle est définie une bonne fois pour toutes par programme, ou elle peut être logique lorsqu'on fabrique, à la volée, des ensembles d'objets ayant des caractéristiques communes.

Clé

Dans une base de données, la même information est parfois dupliquée ou croisée. La notion de clé est dès lors un mécanisme permettant de connaître la donnée de référence et de faire la différence avec celles qui sont simplement des copies de cette donnée de référence. Une clé peut être composée de plusieurs données qui, réunies, forment un identifiant unique d'une chose. Le mécanisme des clés existe tant pour les bases de données relationnelles que XML (on peut définir des clés avec XML Schema).

Clé artificielle

Clé ou élément de clé n'étant pas une donnée propre de l'application mais un identifiant quelconque attribué de manière mécanique à un ensemble de données. Par exemple, un simple numéro incrémental permet d'identifier les rangées d'un tableau, et ainsi de les différencier.

Clé étrangère

Une clé étrangère est tout simplement une information servant de clé ou partie de clé, mais reprenant la valeur d'une clé stockée ailleurs. Le principe des clés étrangères existe également dans XML Schema.

Composant ou constructeur

Les composants sont les représentations concrètes des fonctions définies dans XML Schema : il s'agit des éléments du vocabulaire de XML Schema utilisés par un auteur qui écrit un schéma XML. Remarquez que le terme « composant » ne désigne pas les attributs de ces éléments mais seulement les éléments.

Composition

Terme issu du monde UML, une composition est une association de type « agrégation » où l'existence des objets formant les parties est liée à celle de l'objet formant le tout. La suppression de l'objet parent – le tout – entraîne celle des objets reliés par l'association de composition – les parties.

Conceptuel

On utilise le mot conceptuel dans les expressions *modèle conceptuel*, *modèle conceptuel de données*. Il désigne la représentation idéale de la chose que l'on cherche à maîtriser, l'idée que l'on cherche à transmettre, les données que l'on souhaite organiser, et les traitements et programmes qui feront les contrôles et manipulation des données. En informatique, le modèle conceptuel des données est à un programme ce que le plan est à un immeuble.

Contrôle lexical

Le contrôle lexical consiste à vérifier si les caractères significatifs du langage XML sont utilisés correctement. Par exemple, une valeur d'attribut s'écrit entre deux guillemets droits " et non entre chevrons «.

Contrôler que les caractères utilisés dans les chaînes de caractères des noms, valeurs et données du document XML sont conformes aux définitions lexicales de leurs types fait partie du contrôle lexical. Par exemple, un nom d'élément doit commencer par une lettre, un souligné ou un deux-points, et tout autre caractère est interdit. Ainsi, des noms d'éléments lexicalement valides sont `para`, _para et `:para`, et des exemples de noms invalides sont `1para`, `#para`, etc. Vous pouvez remarquer qu'il est inutile de disposer d'un schéma pour contrôler la validité lexicale de la syntaxe d'un document XML. En revanche, le schéma est impératif pour contrôler la validité lexicale des contenus d'éléments et d'attributs.

Contrôle grammatical

Le contrôle grammatical porte sur la bonne utilisation du vocabulaire : dans l'ordre et à bon escient.

On distingue deux niveaux de contrôle grammatical :

- L'un consiste à contrôler que le schéma est conforme au langage d'écriture de schémas utilisé. Par exemple, en XML Schema, le contrôle grammatical doit vérifier qu'une définition d'élément (constructeur xs:element) n'utilise pas simultanément les attributs `name` et `ref`.
- L'autre consiste à contrôler que les éléments et attributs d'un document XML respectent les règles définies par le schéma dont dépend le document XML. Par exemple, l'élément suivant un `titre` doit être un `para` qui doit lui-même être qualifié par un attribut `date`.

Contrôle syntaxique

La syntaxe désigne l'ensemble des caractères et mots significatifs du langage. La racine grecque du mot syntaxe, *suntaxis*, veut dire « ranger ensemble » (*sun*,

ensemble, et *tassein*, ranger). La syntaxe est donc la partie de la grammaire qui traite du rôle et de la disposition des mots et propositions dans une phrase.

De même qu'en français, le point, la virgule et les autres signes de ponctuation jouent indéniablement un rôle dans l'expression écrite – ils sont remplacés par les intonations dans l'expression orale –, les documents XML et leurs schémas utilisent des caractères significatifs qui leur sont propres. Cela se remarque immédiatement en observant les extraits de ces langages représentant une même définition de modèle :

En XML 1.0 :

```
<!ELEMENT schema ((%include; | %import; | %redefine; | %annotation;)*,
((%simpleType; | %complexType; | %element; | %attribute; |
%attributeGroup; | %group; | %notation; ),
 (%annotation;)*)* )>
<!ATTLIST schema targetNamespace CDATA #IMPLIED >
```

En XML Schema :

```
<xs:element name="schema">
   <xs:complexType>
    <xs:complexContent>
    <xs:extension base="xs:openAttrs">
     <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
       <xs:element ref="xs:include"/>
       <xs:element ref="xs:import"/>
       …
      </xs:choice>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
       <xs:group ref="xs:schemaTop"/>
       <xs:element ref="xs:annotation" minOccurs="0"
                   maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:sequence>
     <xs:attribute name="targetNamespace" type="xs:anyURI"/>
     …
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>
   ….
</xs:element>
```

Les DTD comportent un grand nombre de caractères syntaxiques que les schémas XML remplacent par du vocabulaire. Les caractères significatifs des DTD, notam-

ment les connecteurs, tels que parenthèses, virgules, barres verticales, etc. et les symboles d'occurrences (point d'interrogation, astérisque), sont remplacés dans les schémas par des noms d'éléments. Ces changements résultent de la volonté d'utiliser dans XML Schema la syntaxe XML. Un schéma s'écrit comme n'importe quel document XML tandis qu'une DTD a son propre langage.

Crochet

Un élément crochet est un élément XML conçu de telle sorte qu'il peut servir de point de raccordement d'un schéma XML à un autre (voir aussi anneau).

Data-module code

C'est le code d'identification d'un module de données. Cette codification obéit à des règles strictes qui permettent de donner un sens métier à ce code.

Dérivation

La dérivation est une opération au moyen de laquelle on crée un type à partir d'un autre. Il s'agit d'un terme utilisé dans le monde des schémas XML. Comme des objets, les types dérivés les uns des autres forment un chaînage qui possède ses propres contraintes de fabrication et d'utilisation.

Diagramme de classes

Le diagramme de classes fait partie de la terminologie UML et désigne la représentation graphique des classes d'un système de données et les relations (et cardinalités) qui les relient les unes aux autres. Le diagramme des classes est celui qui se rapproche le plus des besoins de représentation des schémas XML. Il n'est malheureusement pas parfait pour cet usage.

Document électronique

On appel document électronique tout objet informatique pouvant être apparenté à un document, au sens traditionnel du terme : un document papier, une photo, une vidéo, une séquence sonore, peuvent être des documents à partir du moment où ils sont porteurs de sens.

L'informatique offre désormais des zones d'ombre : un courrier électronique, un fichier PréAO (Présentation assistée par ordinateur), un objet multimédia, un fichier CAO (Conception assistée par ordinateur), un échange par messagerie instantanée, sont-ils des documents ? La réponse est affirmative si on considère que ces objets prennent du sens par rapport à un événement particulier. Par exemple, on peut considérer que le courriel historique de John Bosak du 22 novembre 1996 à 14 h 06'58" annonçant la naissance de XML est un document. On peut également considérer

que tout fichier qui ne contient pas des données (au sens d'un alignement de valeurs atomiques) est un document.

Document papier

Un document papier a la caractéristique d'avoir une dimension physique finie : par exemple, une feuille de papier A4 et des contraintes de mise en page (les en-têtes et pieds de page par exemple). Aussi les deux caractéristiques principales d'un document papier sont-elles sa dimension et la typographie qui y est utilisée, parfois de manière très sophistiquée.

Que le document soit papier, ou « papier électronique » comme PDF, ne change pas le problème : pour que l'information soit compréhensible, claire et facile à lire, il faut qu'elle soit composée. Cela fait une grande différence avec les documents dématérialisés tels que des pages HTML pour lesquelles une qualité très moyenne est souvent suffisante (on parle ici du cas de l'information technique et non des pages HTML à caractère marketing où le graphisme est souvent très élaboré).

Document structuré vs non structuré

Autrefois, tout document était qualifié de « *non structuré* » par opposition aux données, souvent stockées dans des tables, qui, de leur côté, étaient qualifiées de « *structurées* ».

Avec XML, ce distinguo n'a plus lieu d'être : un document textuel est parfaitement structuré.

Document textuel vs données

Une donnée est une valeur atomique qui n'a pas de sens en soi. Un tableau de chiffres (des dates par exemple) n'a jamais donné de sens à ces données. En revanche, il est possible de manipuler ces données (faire des calculs, des extractions, des tris, etc.).

À l'opposé, un document est a priori un objet permettant de donner de la valeur aux données. Ce faisant, et avant que XML n'arrive, il n'était plus possible de manipuler les données.

Un document XML a dès lors l'avantage de pouvoir contenir indifféremment des données et/ou des textes sur lesquels les manipulations et transformations sont toujours possibles. On est alors amené à parler de « données données » et « données textuelles ».

Document XML

Est baptisé document XML tout fichier contenant des données balisées XML respectant les règles de bonne forme. Le mot document n'est donc pas ici à prendre dans son sens commun.

Documentation modulaire structurée

L'expression de documentation modulaire structurée désigne une méthode de conception de la structure d'un document (ou de tout autre objet) par laquelle des modules représentent des composants textuels ou graphiques élémentaires. Les modules contiennent des données balisées et ils sont assemblés entre eux selon un ordonnancement logique, lui-même défini au moyen d'un modèle XML.

DTD

Le mot DTD est l'acronyme de deux choses différentes : d'une part *définition de type de document* et d'autre part *déclaration de type de document*.

Remarquez que, dans les deux cas, il faut dire « la » DTD et non « le » DTD comme on le voit trop souvent écrit.

La définition de type de document est le fichier qui contient des définitions d'éléments, d'attributs, d'entités et de notations. C'est en général ce fichier qui est désigné quand on parle de « *la DTD du document XML* ».

La déclaration de type de document est la carte que l'on trouve dans le prologue d'un document XML et qui s'écrit : `<!DOCTYPE toto SYSTEM "toto.dtd">`. Cette déclaration spécifie l'élément racine du document et lui associe la « DTD » qui convient.

Effectivité

L'effectivité complète les notions d'applicabilité. C'est une information qui indique le niveau d'obsolescence d'une autre information, en particulier d'une publication. Par exemple, un module d'information peut avoir une applicabilité aux systèmes A, B et C, mais peut faire partie d'une publication dédiée, quant à elle, au seul système B. Le `uplet {A,B,C}` représente l'applicabilité du module tandis que le singleton `{B}` représente l'effectivité de la publication.

Élément de données

Un élément de données ne contient que des données (numériques ou textuelles peu importe) ou un modèle qui peut être assimilé à de la donnée (cas d'un élément mixte par exemple).

Élément de données global

Un élément de données global contient une collection d'éléments de données, potentiellement mêlés à des données textuelles, voire des éléments purement structurels.

Élément flottant

Un élément est dit flottant quand on peut le mettre n'importe où dans une structure. C'était possible avec SGML grâce au mécanisme des exceptions ; cela ne l'est plus avec la version actuelle de XML. Le mouvement s'inversera peut-être un jour.

Élément global

Il s'agit d'une terminologie XML Schema qui désigne la qualité d'un élément à être utilisé comme élément racine d'un document XML et, par référence, dans plusieurs autres éléments XML (voir également élément local ci-après).

Élément local

Il s'agit d'une terminologie XML Schema désignant un élément qui ne peut être utilisé ni comme racine d'un document XML ni par référence à l'intérieur de plusieurs autres éléments. Un élément XML local n'appartient qu'à un seul élément XML parent (voir également élément global ci-avant).

Élément mixte

Un élément mixte contient, sur le même niveau hiérarchique, un contenu textuel et des sous-éléments. Un tel élément est typique de documents XML représentant des documents papier.

Élément purement structurel

Un élément purement structurel ne contient aucun élément de données comme enfant direct.

Élément racine

Un élément racine est le premier élément d'un document XML. Un fichier renfermant un document XML ne peut contenir qu'un seul élément racine.

Ensemble d'informations

Un ensemble d'informations est tout simplement un document XML. Cette locution, qui est la traduction de l'anglais *infoset*, est en train de rentrer dans le vocabulaire courant de XML et permet avantageusement d'éviter d'utiliser le mot *document*, parfois trop proche des notions de document papier.

Ensemble d'informations bien formé

Un ensemble d'informations bien formé, ou *well-formed infoset*, est un ensemble d'informations qui vérifie les critères de bonne forme de XML.

Entité

Dans le vocabulaire de XML, une entité est une ressource informatique que l'on peut spécifier à l'aide d'une déclaration d'entité. Une entité est typiquement soit une chaîne de caractères, soit un fichier XML, ASCII ou binaire.

Espaces de noms, espaces de nommage

Les espaces de noms, ou encore *de nommage*, servent à définir les limites de validité des vocabulaires définis par les schémas XML. Ainsi, une même balise `<prix>` peut avoir deux significations différentes selon qu'elle appartient à un espace de noms `A` ou `B`, et on écrira pour les distinguer `<a:prix>` et `<b:prix>`.

Feuille

Une feuille est un élément terminal d'une arborescence de données XML, ou arbre XML. À ne pas confondre avec une feuille de papier !

Forme

La forme est l'allure que prend un ensemble de données lorsqu'il est écrit en utilisant une syntaxe particulière. La forme XML se reconnaît aisément grâce à ses balises et attributs emboîtés les uns dans les autres.

Forme canonique

La forme canonique d'un document XML enlève les ambiguïtés résultant des différentes formes lexicales que peut avoir un document XML, et ce afin de pouvoir simplifier les applications de traitement des documents XML et même comparer de tels documents deux à deux. Les parseurs et les validateurs sont censés être capables de produire les formes canoniques des documents qu'on leur soumet.

Voici les caractéristiques d'une forme canonique :

- utilisation du seul codage UTF-8 ;
- normalisation des valeurs d'attributs et repositionnement des attributs dans l'ordre alphabétique de l'initiale de leur nom ;
- suppression du prologue du document ;
- changement de notation des éléments vides, écriture de `<gi></gi>` au lieu de `<gi/>` ;
- intégration des sous-documents : les appels d'entités externes analysées sont remplacés par les entités désignées ;
- homogénéisation des déclarations de préfixes d'espaces de noms ;
- suppression des préfixes d'espaces de noms des éléments et attributs où ils sont inutiles ;
- inclusion des attributs par défaut.

Toutes les questions relatives à la canonisation ne sont pas réglées et ce sujet reste en discussion.

Forme lexicale et forme lexicale canonique

La forme lexicale est celle que l'on peut voir en consultant un document XML avec un éditeur ASCII, donc tel qu'il est enregistré sur disque, sans interprétation aucune des balises. La forme lexicale du chiffre 3 peut être `"03.00"` ou `"3"` ou `" 3 "`, cela ne change pas la valeur numérique de ce contenu. XML Schema spécifie des formes lexicales canoniques pour chaque type (dates, durées, nombres entiers, flottants, chaînes de caractères, etc.).

Gestion de configuration

La gestion de configuration consiste à gérer la version de chaque composant entrant dans la conception d'un système. Dans le cas de publications, il s'agit d'identifier chaque objet entrant dans la composition du document, qu'il s'agisse d'éléments textuels ou graphiques.

Gestion de contenu

Cela concerne la gestion des composants entrant dans la composition des pages web. Un outil de gestion de contenu permet de modéliser les pages d'un site web, produire les contenus textuels, gérer les liens entre les pages et avec les illustrations, et surtout les règles d'affichage en fonction de profils d'utilisateurs ou d'événements autres.

Gestion de l'applicabilité

Voir applicabilité.

Gestion de versions

Cela consiste à gérer des variantes d'une version de base. Cela peut se faire par recopie des fichiers ou par modification à l'intérieur du document XML. Un système doit permettre de conserver l'historique des liens avec le fichier d'origine, et éventuellement de provoquer des alertes quand le fichier d'origine est modifié.

Gestion des révisions

La gestion des révisions consiste à stocker l'historique des fichiers. La gestion des révisions comprend toujours deux dimensions : d'une part une gestion des éditions (un numéro s'incrémente à chaque fois qu'une modification mineure est effectuée) et d'autre part la gestion des révisions (après une série d'éditions, une version finale est créée qui consolide toutes les éditions).

Gestion des connaissances

La gestion des connaissances (*Knowledge Management* en anglais) consiste à gérer des documents, des informations sur ces documents et une partie de l'activité qui accompagne un domaine particulier. Ainsi, si des forums de discussion sont ouverts sur des thèmes particuliers, il est intéressant d'archiver les propos échangés pour les retrouver un jour, et parfois même très longtemps après. Les savoir-faire, associés à des profils d'utilisateurs, sont rangés dans des classeurs thématiques virtuels auxquels on accède de plusieurs manières.

Gestion électronique de documents

La gestion électronique des documents concerne les systèmes dédiés au contrôle des cycles de production des documents. Plus largement, la GED concerne tout système destiné à contrôler un référentiel de documents. Pour cela, les documents sont accompagnés de métadonnées qui servent à les classer, les identifier, et fournir un résumé ou des mots-clés relatifs à leur contenu.

Grammaire

Une grammaire est constituée des règles de positionnement des noms d'éléments et d'attributs les uns par rapport aux autres. Un schéma XML définit un vocabulaire (ensemble des noms des éléments et d'attributs) et une grammaire.

Comme dans une langue naturelle, en XML la grammaire sert à définir la logique d'assemblage des mots. Chaque schéma définit une logique d'assemblage des éléments et des attributs.

Héritage

L'héritage est un concept qui vient du monde des objets. Ces derniers font partie de classes qui sont toujours créées à partir de classes parents. Ainsi, les propriétés et méthodes d'une classe sont pour partie héritées de celles de leurs parents.

Héritage et dérivation

Quand on veut modifier les caractéristiques, ou les méthodes, héritées d'une classe parent, on opère ce qui s'appelle une dérivation.

Identifiant, identificateur

L'identifiant d'un élément est ce qui permet de l'identifier de manière unique : il s'agit d'une clé simple composée d'une seule chaîne de caractères. Avec XML 1.0, le seul type d'identifiant autorisé était le type `ID`. XML Schema permet de définir, quant à lui, de vraies clés qui sont des `uplets` prenant la forme `{A,B,C…}` (voir le mot *clé* dans ce glossaire).

Infrastructure UML 2.0

Il s'agit du premier livre de la spécification UML 2.0. Il définit les principes fondateurs servant de base à la création de tous les modèles de UML 2.0.

Voir superstructure UML 2.0.

Illustration code number

Comme le code d'identification des modules (voir *data-module code*), celui destiné aux illustrations est un identificateur unique pour une application métier. À la différence d'un module, ce code ne peut pas être inscrit dans le fichier lui-même et doit obligatoirement correspondre à un nom de fichier ; il ne peut donc s'agir que d'une chaîne de caractères.

Infoset

Un infoset, littéralement *ensemble d'informations*, fait l'objet de la recommandation du W3C XML Infoset datée du 24 octobre 2001. Il s'agit de la nouvelle terminologie utilisée pour désigner un document XML. Cette recommandation a un rôle très important. Grâce à elle, les spécifications du modèle de documents XML et des schémas qui servent à les contrôler sont, pour la première fois, radicalement séparées. En effet, la norme ISO 8879 (SGML) et la recommandation XML 1.0 du W3C mêlaient, jusqu'alors, la définition du modèle de document XML et celle du langage d'écriture de leurs schémas : les DTD.

XML Infoset ne définit aucun langage d'écriture de schéma. Cette recommandation spécifie qu'un document XML est un ensemble d'informations composé d'unités d'information et de contenus textuels.

Nous avons choisi dans cet ouvrage de ne pas utiliser les expressions *infoset* et *ensemble d'information*, mais celle de *document XML*.

Si l'on sait habituellement qu'un document XML est composé des unités d'information de type élément et attribut, on sait moins en revanche qu'il peut contenir jusqu'à neuf autres types d'unités d'information, soit onze au total. L'objectif de XML Infoset est de donner une définition précise de chacun d'eux.

Ces onze types d'unités d'information sont les suivants :

- la déclaration de type de document ;
- le document, c'est-à-dire l'élément racine du document XML ;
- l'élément ;
- l'attribut ;
- l'espace de noms ;
- l'instruction de traitement ;

- le commentaire ;
- la référence à une entité externe ;
- l'entité externe ;
- le caractère ;
- la notation.

Les logiciels et textes normatifs qui se disent conformes à XML Infoset doivent prendre en charge la totalité de ces onze types d'unités d'information.

Les langages d'écriture de schémas de documents XML ne concernent qu'une partie de ces unités d'information : celles qui font la structure et le vocabulaire d'un document XML – les éléments, les attributs, les espaces de noms, et dans une moindre mesure les notations.

Sur le seul point de vue logique, référencer les entités externes utilisées par un document XML à partir de son prologue, c'est-à-dire dans un complément à la DTD, est une anomalie de conception de ces langages que les nouvelles approches, XML Schema et RelaxNG, corrigent avec XLink et XInclude.

A contrario, des informations définies dans les schémas n'apparaissent pas en tant qu'unités d'information dans XML Infoset. Il s'agit par exemple des types de données. Le fait qu'une chaîne de caractères soit un nombre entier, une date ou une forme lexicale particulière n'est pas une information censée être contenue dans le document XML lui-même.

Ainsi, il est aujourd'hui flagrant qu'un document XML et son schéma sont deux sources d'information indépendantes et complémentaires.

Voilà pourquoi le concept de PSVI, *Post Schema Validation Infoset*, a été inventé. Il s'agit d'un document XML qui contient l'infoset de départ, canonisé et augmenté des informations issues de son schéma. Ainsi, le PSVI contient les informations sur le type des données, les modèles auxquels appartiennent les éléments, la nature des attributs (imposé, obligatoire, par défaut…). Le PSVI peut être considéré comme une sorte de fichier XML exporté dans lequel on aurait à la fois le document XML d'origine et la totalité des informations issues de son schéma et le concernant. Le PSVI est un cas particulier d'ensemble d'informations.

Instance

Une instance est un objet créé à partir d'un modèle et conforme à ce dernier. Dans le cas de SGML, les instances sont les documents SGML conformes à une DTD mais, en XML, le mot instance a disparu au profit des expressions *document XML valide* et *ensemble d'informations XML valide*. En ce qui concerne UML, et le monde des objets plus généralement, l'instance (d'une classe) est un objet qui possède les comportements de la classe à laquelle il appartient. Cet objet est alors le composant

effectif des programmes (c'est lui qui calcule) alors que la classe est plutôt une définition ou une spécification de l'ensemble des instances à venir.

À notre grand étonnement, nous avons appris que ce mot a été refusé à plusieurs reprises par l'Académie française, malgré son usage répandu parmi les informaticiens et sa présence dans le grand dictionnaire terminologique québécois de la langue française (http://www.olf.gouv.qc.ca/ressources/gdt.html).

Langage de liaison

Un langage de liaison sert à écrire (établir) des liens ; il s'agit typiquement de XLink.

Lien entre documents vs intra-document

Un lien entre documents part d'un document XML pour aller dans un autre tandis qu'un lien intra-document reste limité au document XML où se trouve la source du lien. Dans ce deuxième cas, un simple identifiant suffit, tandis que dans le premier il faut ajouter, au minimum, le chemin d'accès du deuxième document. On notera enfin que dans le cas du lien entre documents, le document dans lequel se trouve la cible du lien devient subordonné au premier : on ne peut le déplacer sans intervenir sur celui où se trouve la source.

Liste des pages effectives

Une liste effective des pages est, en quelque sorte, une table des matières des pages d'une publication. Concept réservé aux documents techniques sensibles, la liste effective des pages permet de lister chaque page de l'ouvrage avec, en regard, son indice de révision et son statut (effective ou détruite). Une telle liste est l'un des moyens de contrôle des révisions d'une publication.

Métadonnées

Une métadonnée est une donnée sur une donnée : par exemple, spécifier que le nombre 1590575118235 est un numéro de sécurité sociale nécessite de faire appel à une métadonnée (en l'occurrence celle qui précisera le type de ce nombre). Dans le monde des documents, les métadonnées sont toutes les informations nécessaires à la gestion administrative dudit document (numéro ISBN, noms des auteurs, dates de parution, etc.).

Métalangage

Un métalangage est un langage qui sert à écrire d'autres langages. Si on considère que le vocabulaire (noms des éléments, des attributs et de leurs valeurs possibles) défini par une DTD ou un schéma XML est un langage, alors le langage utilisé pour écrire de tels modèles est un métalangage. La syntaxe des DTD pouvant être modi-

fiée, SGML les définissait à l'aide d'une syntaxe abstraite ; on avait vraiment là un métalangage.

Métamodèle

Un métamodèle est le père d'un modèle, on dit aussi le modèle d'un modèle. Le terme de méta fait référence au niveau d'abstraction supplémentaire induit naturellement par la position de « père de modèle ». Pour XML, il existe deux branches parentes : celle de la syntaxe et celle des schémas.

Dans la branche de la syntaxe, le modèle père d'un document XML est celui qui définit la syntaxe concrète d'écriture des documents XML (en l'occurrence la recommandation Infoset), tandis que le métamodèle (le grand-père) est celui qui définit les relations parent-enfant, le fait que les éléments aient des propriétés (les attributs), la notion de prologue, etc. Ce métamodèle est visible dans la norme SGML ISO 8879:1986.

Dans la branche des schémas, il existe plusieurs modèles pères (XML Schema, RelaxNG et Schematron) et plusieurs métamodèles qui sont les modèles conceptuels ayant servi à définir les vocabulaires (et les sémantiques associées) de chacun de ces langages de modélisation. XML Schema et RelaxNG ont le même métamodèle : celui des langages à base de grammaire. Schematron est le résultat d'un métamodèle différent : celui des langages à base de règles.

En valeur absolue, tout modèle servant à produire d'autres modèles est un métamodèle.

Modèle

Le mot modèle a deux sens. Il peut désigner le moule qui permet de reproduire à l'identique une même chose comme il peut désigner une représentation simplifiée d'un ensemble de choses. Par exemple : un cylindre marron, des racines, une boule verte et des points rouges à l'intérieur représentent tous les pommiers de la terre ; un cercle jaune et des rayons : c'est le soleil.

Modèle et schéma conceptuels

Il existe deux types de modèles conceptuels. L'un concerne les langages d'écriture de schémas, l'autre les données.

En ce qui concerne les langages, un modèle conceptuel est la définition théorique du langage, soit le préalable nécessaire à la définition d'une forme concrète du langage. Un langage – par exemple le langage d'écriture de schémas XML du W3C – doit être défini de manière théorique avant d'exister sous une forme concrète, ou syntaxique. XML Schema est défini d'une part par des règles de production et d'autre part par des explications en langage naturel. Quant à lui, RelaxNG est formellement défini par des formules mathématiques dites règles d'inférence.

En ce qui concerne les données, un modèle conceptuel permet de les représenter ainsi que les relations qui les unissent indépendamment de la manière selon laquelle elles seront physiquement organisées, et notamment stockées. Un schéma conceptuel est dès lors une application d'un modèle conceptuel de données pour une application particulière. UML propose un modèle conceptuel de représentation des données. Un diagramme de classes UML réalisé pour une application particulière représentera le schéma conceptuel des données de cette application.

Modèle logique

Un modèle logique représente la manière selon laquelle les données seront réellement mises en œuvre pour une application. Un schéma XML, une DTD, sont des modèles logiques : ils représentent la manière dont seront réellement organisées les données, et notamment la manière dont elles seront stockées les unes par rapport aux autres.

Par exemple, on peut avoir un modèle conceptuel représentant des familles composées de parents et d'enfants : la présence d'une relation parent-enfant est *de facto* évidente. Ce modèle conceptuel ne signifie pas pour autant que les éléments XML correspondants seront `<famille>`, `<père>`, `<mère>`, `<fille>` et `<fils>`, et que ces derniers seront des enfants des éléments `<père>` et `<mère>`. Le modèle logique retenu *in fine* pour représenter les familles pourra être totalement différent des sous-entendus du modèle conceptuel. On pourrait même très bien avoir des éléments `<individus>` et `<relation>` permettant de lier les individus entre eux selon une logique de relations familiales.

Modèle vs schéma, langage de modélisation

Rick Jelliffe, auteur de la spécification Schematron, fait une distinction entre les termes « schémas » et « modèles ». Selon lui, un schéma définit de manière exhaustive et complète une réalité tandis qu'un modèle n'en est que la meilleure représentation possible.

Nous soutenons cette approche que nous expliquons par une analogie : quand un homme sert de *modèle* aux autres, il est un point de repère, un guide, un exemple à suivre, mais quand on affirme suivre le *schéma* de pensée d'une personne, c'est qu'on déroule à l'identique un même scénario intellectuel. De même que le sculpteur dispose d'un modèle et d'une technique pour réaliser son œuvre, les modèles et les schémas sont deux notions nécessaires et complémentaires pour produire un document XML.

Dans le domaine technique, on peut dire d'une voiture qu'elle est de type break quand on peut la comparer à un objet de référence qui a ce type : un autre break montré en exemple. Cet objet de référence est le modèle qui nous sert à reconnaître

le type d'une autre voiture. La définition exacte de chaque voiture est cependant constituée des dessins techniques qui ont servi à la fabriquer : les plans, ou schémas.

Le mot schéma est donc à prendre dans le sens que lui donnent les Américains, celui de plan très précis d'un objet. Nous avons suivi ce *distinguo* dans cet ouvrage et avons choisi d'utiliser la terminologie suivante :

- Le mot *schéma* est utilisé pour désigner l'ensemble des règles définissant une classe de document. Un schéma est typiquement une DTD ou un fichier contenant un document XML Schema ou RelaxNG.
- Le mot *modèle* désigne la chose, ou l'idée, que l'on veut représenter. Cela est très différent du schéma. Il en est ainsi des idées ayant prévalu à la définition concrète des schémas et des documents XML. En effet, tout document XML peut avoir un modèle, à savoir un document XML exemple montrant ce qu'est la syntaxe XML et mettant en scène des éléments, des attributs et d'autres unités d'informations. Toutefois, seul un schéma permettra de savoir si un document XML en particulier est bon ou pas pour une application spécifique.
  Il existe toutefois des exceptions « historiques », telle l'expression « modèle de contenu d'un élément » qui désigne, dans une DTD, la définition d'un élément. Dans un schéma XML, le vocabulaire est plus précis, et on préférera parler du type de l'élément.
- L'expression *langage de modélisation* n'est pas utilisée ; nous lui préférons *langage d'écriture de schémas*. De la même manière, l'expression *langage de schématisation* n'est pas utilisée : elle choque sans correspondre à la réalité qu'on voudrait lui donner. En ce qui concerne les modèles, nous ne parlerons que de représentation des modèles.
- Le mot métamodèle est utilisé pour désigner le modèle du modèle, il désigne bien souvent les concepts purement intellectuels ayant prévalu à la définition d'un modèle. Il existe par exemple un métamodèle pour XML qui est constitué du concept d'objet (les types), de propriétés (les attributs) et d'héritage (les relations parent-enfant créées par les emboîtements des éléments forment des lignées d'héritages sémantiques par le seul jeu des noms d'éléments).

Module

Dans le présent ouvrage, un module est un document XML destiné à faire partie d'un ensemble plus vaste. Le module n'a pas de taille prédéfinie, il peut s'agir d'un paragraphe comme d'un chapitre entier : c'est son auteur qui la détermine. Un module est géré à l'aide de métadonnées.

Nœud

Un nœud est, dans un arbre représentant une structure XML, un embranchement.

Objet métier

Dans cet ouvrage, un objet métier est un élément XML dont le nom est directement interprété par les programmes de traitement de l'information. Par exemple, si l'élément `<dateDeValeur>` est un objet métier, il sera associé à un programme de même nom.

Occurrence et nombre d'occurrences

L'occurrence est la répétition d'une même chose. Aussi, le *nombre d'occurrences* est le nombre de fois qu'une chose apparaît. En XML, la notion d'occurrence intervient dans les modèles (DTD et schémas) car c'est là que le concepteur précise combien de fois un élément est autorisé ; on parle alors du *nombre d'occurrences*. En UML, ce *nombre d'occurrences* est représenté par la multiplicité.

Ontologie et thésaurus

Le dictionnaire Larousse donne de ces mots les définitions suivantes :

Ontologie : (gr. *On, ontos*, être, et *logos*, science). Discours issu de la logique mathématique et de la linguistique, qui traite des termes utilisés pour désigner les êtres constitutifs de la réalité.

Thésaurus : (gr. *Thesauros*, trésor). Dictionnaire destiné à aider les recherches dans certaines disciplines et contenant, pour chaque mot-clé, les termes similaires ou synonymes.

Une ontologie est un dictionnaire de mots doublés de définitions de relations entre eux. À la différence d'un dictionnaire normal dont le seul sens de lecture est le sens alphabétique, les relations permettent à une ontologie d'avoir plusieurs sens de lecture.

De son côté, un thésaurus est comparable à une ontologie réduite à un seul domaine et un seul type de relation entre les mots : la relation (hiérarchique) d'appartenance à un domaine. Un thésaurus est donc un dictionnaire augmenté d'une seule structure additionnelle : une classification hiérarchique des mots par domaine.

Parseur (voir validateur)

PSVI (voir la section Infoset)

Persistance

La persistance s'applique aux données qui doivent être stockées pour « persister » au-delà du temps d'exécution d'un programme. Dans les applications non standardisées, la persistance pose simultanément le problème du format de stockage des données (des champs de fichiers ASCII séparés par des tabulations ? des champs de base de données ?, etc.) et de leur lieu de stockage. Cependant, avec XML, la question du format de stockage ne se pose plus (car les normes relatives à XML sont très précises

sur la question). Ne subsiste que la question du lieu de stockage du document XML (dans des arborescences de fichiers, au sein d'une base de données XML, etc.), et éventuellement du vocabulaire qui sera utilisé.

Polymorphe

Littéralement, se dit d'un objet qui présente plusieurs formes. Dans le monde XML, un objet est dit polymorphe lorsque le sens de la donnée qu'il contient dépend du contexte dans lequel l'objet est baigné. Par exemple, le fragment `<date>2004-08-27</date>` prend un sens différent selon que l'élément parent est `<Paiement>` ou `<Facturation>`, par exemple. Aussi, dans la conception de schémas XML, le choix des noms attribués aux éléments XML peut ne pas être anodin. Un élément qui s'appellerait `<dateDePaiement>` serait moins polymorphe car il garderait une bonne partie de son sens même isolé de son contexte.

Polymorphisme

Le polymorphisme concerne les messages envoyés aux objets : un même message peut être envoyé à différents objets qui, en fonction de leur type, rendront des réponses différentes. En XML, le polymorphisme concerne les éléments ayant un même nom mais des types différents (c'est possible avec XML Schema).

Publication module code

Le code d'un module de publication est l'identifiant unique attribué à la structure d'assemblage d'une publication. En général, cet identifiant est également visible dans la publication finale.

Référence

Une référence est un identifiant permettant de connaître la cible d'un lien.

Référence d'entité

En XML, une référence d'entité (on devrait dire *référence à une entité*) est un identifiant renvoyant à une déclaration d'entité.

Réification

La réification est un concept relatif aux cartes de topiques. C'est l'instanciation physique, concrète, d'une abstraction, en particulier un sujet. Quand on met un sujet dans un certain contexte, il devient un topique : on dit que le sujet est réifié.

Ressource

Une ressource est toute information électronique accessible à partir d'une adresse URL. À ne pas confondre avec les mémoires d'ordinateur, imprimantes et autres systèmes physiques également appelés ressources dans le monde informatique.

Schéma conceptuel

Voir modèle conceptuel.

Schéma à base de grammaire

Un schéma à base de grammaire définit la syntaxe d'un vocabulaire XML en totalité : les noms des éléments, des attributs, leur type, parfois leur contenu imposé. Contrairement aux schémas à base de règles, ces schémas ne permettent pas aux documents XML d'être partiellement valides.

Voir aussi la rubrique *Métamodèle*.

Schéma à base de règles

Un schéma à base de règles définit des règles que le document XML doit vérifier et qui peuvent tout aussi bien porter sur la présence (un certain nombre de fois) d'un élément que sur une corrélation entre la valeur d'un attribut et le contenu d'un élément. Contrairement aux schémas à base de grammaire, ces schémas permettent d'avoir des documents XML « partiellement » valides. Expliquons ce que l'on entend par partiellement valides : certes, toutes les règles édictées par le schéma doivent être vérifiées, mais il n'est pas obligé que ces règles portent sur tous les éléments du document XML.

Voir aussi rubrique *Métamodèle*.

Schémas

Voir la rubrique *Modèle vs schéma*, *langage de modélisation*.

Schematron

Langage d'écriture de modèle XML à base de règles. Les règles de Schematron sont écrites principalement en s'appuyant sur le langage Xpath.

Sémantique

La sémantique est le sens profond d'un nom, par rapport à son contexte d'utilisation. La sémantique des éléments, de leurs attributs, représente le sens qu'ils ont par rapport à l'application qui les utilise. Plus la sémantique est précise et moins l'élément est polymorphe.

Sérialisation

Le mot sérialisation désigne la transformation qui consiste à produire un fichier XML physique à partir de sa représentation en mémoire. XML est tel qu'il n'existe aujourd'hui pas une seule représentation possible mais plusieurs. La sérialisation consiste à reproduire physiquement dans un fichier le sens de lecture d'un document XML en conservant les codes permettant de comprendre sa structure arborescente, ainsi que toutes les autres unités d'information autorisées.

La sérialisation est donc la traduction d'un sens de parcours de l'arbre. Nous allons expliquer ce propos à l'aide de trois figures inspirées du livre de Ronald Bourret, *Mapping DTDs to Databases*, paru en 2000 aux éditions O'Reilly.

Il existe quatre manières d'envisager le sens de parcours d'un arbre :

- l'ordre défini par le ciblage des éléments ;
- l'ordre hiérarchique ;
- l'ordre du document ;
- l'ordre des classes d'objets.

Pour les illustrer, nous utiliserons le fragment suivant :

```
<para>IDENTIFICATION BY CUSTOMER<custname>Air Canada</
custname><cus>ACN</cus><xref refid="ibc001"></para>
```

L'ordre de ciblage correspond à la notion de fratrie, qui se définit comme étant l'ensemble des éléments d'un même niveau hiérarchique. Pour le fragment précédent, cet ordre est illustré à la figure E-1.

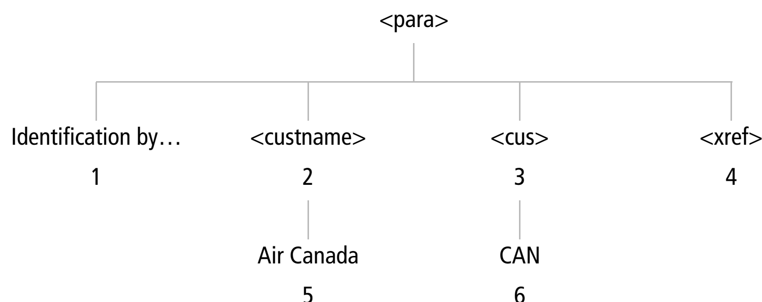


Figure E–1 Représentation graphique de l'ordre des éléments selon la règle du ciblage

L'ordre hiérarchique correspond à la profondeur des éléments dans l'arborescence définie par la structure depuis la racine. Cet ordre est illustré à la figure E-2.

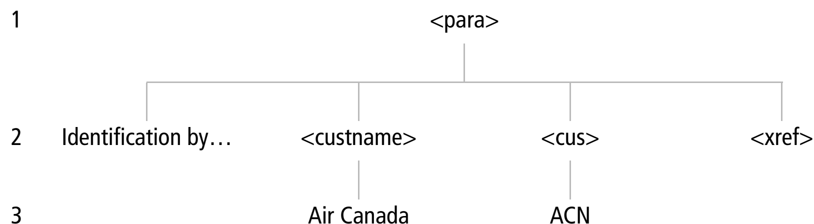


Figure E–2 Représentation graphique de l'ordre des éléments selon la règle hiérarchique

L'ordre du document est celui qui représente l'ordre des éléments dans le sens de la lecture. Cet ordre est illustré à la figure E-3.

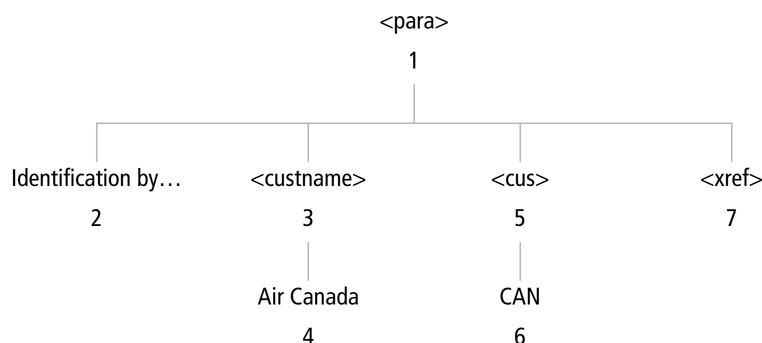


Figure E–3 Représentation graphique de l'ordre des éléments selon la règle du document

Dans les applications orientées objet telles que DOM (Document Object Model), la quatrième manière de parcourir une arborescence d'objets, c'est de cibler les objets d'une même classe. Si les balises `<para>`, `<custname>` et `<cus>` représentent trois sous-classes de la classe des éléments et si `Identification by …`, `Air Canada` et `CAN` représentent trois nœuds de l'ensemble des nœuds textuels, on peut envisager de balayer cet ensemble d'objets en sautant de paragraphe en paragraphe, de lien en lien, de nœud textuel en nœud textuel, etc. Ce mode de lecture est notamment l'une des caractéristiques de XPath. Ce mode de parcours ne fait pas ici l'objet d'une illustration car il ouvre la voie à autant de possibilités qu'il existe de classes d'objets dans un document XML, et car il n'est jamais utilisé dans le but de stocker un document XML.

Sérialisé

Un document XML est sérialisé lorsqu'il a été enregistré physiquement. Quand les données sont stockées en mémoire sous la forme d'un DOM (Document Object Model), la sérialisation va de soi mais quand il s'agit d'objets, la sérialisation est le résultat de la transformation des champs et propriétés publics d'un objet, ou les paramètres et valeurs de retour des méthodes, en un flux XML conforme à un schéma XML spécifique. C'est typiquement ce qui se passe lors des opérations de sérialisation de SOAP.

Serveur de ressources

La notion de serveur de ressource intervient lorsque les liens entre ressources sont définis logiquement, sans utilisation d'adresses physiques. Au moment de la résolution du lien, il faut bien transposer l'identifiant logique en adresse physique. On y procède par programme ou par un serveur de ressources qui a la capacité de recevoir un identifiant logique et de retourner l'URL physique de la ressource correspondante.

Source

Dans notre ouvrage, la source désigne toujours la source d'un lien ; à ne pas confondre donc avec un code source. La source d'un lien est la ressource qui contient le point de départ de ce lien.

Squelette de publication

Un squelette de publication est une ossature écrite en XML qui spécifie les composants d'une publication, leur ordre d'apparition, ainsi que les numéros de révision des composants utilisés.

Structure d'assemblage

Voir squelette de publication.

Superstructure UML 2.0

Il s'agit du deuxième livre de la spécification UML 2.0. Sur les bases des principes fondateurs définis dans le livre *infrastructure UML 2.0*, il définit l'ensemble des concepts de modélisation proposés par UML 2.0.

Voir infrastructure UML 2.0

Syntaxe abstraite

On appelle syntaxe abstraite une syntaxe dans laquelle les caractères spécifiques sont remplacés par des variables. Ainsi, une telle syntaxe laisse la porte ouverte à plusieurs

implémentations. Quand les variables sont remplacées par les caractères définitifs, on obtient une syntaxe concrète.

Syntaxe concrète

Voir syntaxe abstraite.

Système d'information

Un système d'information représente l'ensemble des moyens humains et technique mis en œuvre pour gérer les informations nécessaires au fonctionnement d'une entreprise. Il couvre les grandes fonctions de création, archivage, transformation et diffusion de l'information. À ce titre, il fait appel aux techniques informatiques de gestion des données, des traitements et des communications, ainsi qu'aux méthodes de mise en œuvre de ces techniques.

Taxinomie

Étude théorique des bases, lois, règles, principes d'une classification. Le terme est utilisé pour désigner des classifications de mots.

Synonyme de taxonomie.

Taxonomie

Voir taxinomie.

Topics Map

Voir la rubrique *carte de topique*s.

Topique

Voir la rubrique *carte de topique*s.

Type d'un élément

Le type d'un élément est la définition de la nature de son contenu. En XML, le type peut être simple, l'élément ne contenant qu'une chaîne de caractères, ou complexe quand l'élément renferme des sous éléments et/ou des attributs.

Les chaînes de caractères peuvent elles-mêmes être de plusieurs types : dates, nombres, durées, booléens, etc.

Parmi les types complexes, on trouve les types mixtes (texte et sous-éléments mêlés) et vide (l'élément ne contient rien).

Unité d'information

Une unité d'information est un module.

À ne pas confondre avec une *unité documentaire*, expression parfois utilisée pour désigner un ensemble de modules intermédiaire entre un module et une publication.

Voir la rubrique *module*.

Urbanisation

L'urbanisation est une discipline de gestion des systèmes d'information visant à aligner les fonctions requises par les divisions opérationnelles de l'entreprise avec l'architecture logicielle et technique des systèmes informatiques. L'objectif en est de définir les règles d'équilibre entre la dynamique requise par les processus toujours mouvants de l'entreprise et la nécessaire structuration des données et des programmes propre aux techniques de développements informatiques.

Validation de schéma

La validation de schéma est une opération par laquelle un document XML est contrôlé par rapport au schéma qu'il est censé respecter. Quand le contrôle est positif, il est possible de produire un PSVI, ou Post Schema Validation Infoset, qui est un ensemble d'informations XML contenant la totalité de l'information : le document XML et son schéma associé (attention, un PSVI n'est pas qu'une simple concaténation des deux mais une fusion).

Validateur et parseur

Un validateur est un programme dont le rôle est de contrôler la conformité d'un document XML par rapport à un schéma.

Un parseur – en français analyseur lexico-syntaxique – est un programme qui a pour rôle de vérifier la validité lexicale et syntaxique d'un document XML.

À l'époque où seules les DTD existaient, le parseur contrôlait le vocabulaire, les caractères et les structures utilisés dans un document XML. La tendance actuelle est d'avoir plusieurs types de validation : conformité d'un document XML par rapport à la recommandation XML Infoset, conformité par rapport à un ou plusieurs schéma(s)…

Le « parsing » ou analyse lexico-syntaxique, est le contrôle de la validité lexicale et syntaxique d'un document XML : les caractères utilisés dans le document XML sont-ils autorisés ? la syntaxe XML est-elle respectée ? le document est-il bien formé ?

La validation, c'est le contrôle de la conformité d'un document XML par rapport à un schéma. La validation peut ajouter des informations dans le document XML – par exemple, ajout des valeurs d'attributs par défaut – et en modifier le contenu – par exemple, écriture homogène des nombres et suppression des blancs anormaux.

Validation sémantique

La validation sémantique consiste à contrôler que des éléments, des attributs ou des valeurs de contenus sont utilisés correctement par rapport à leur sens.

La validation sémantique s'applique aux documents XML comme aux langages d'écriture de schémas.

La sémantique peut être définie formellement – c'est le cas de RelaxNG *via* des règles d'inférence –, ou par des explications en langage naturel – c'est le cas des schémas XML du W3C.

Concernant les données, la sémantique peut être contrôlée par rapport à des thésaurus, des ontologies, voire des types.

Vocabulaire (voir aussi grammaire)

Un vocabulaire est l'ensemble des noms d'éléments et d'attributs définis par un schéma XML.

Web sémantique

Le concept de Web sémantique vise à faire prendre conscience de l'importance de la question du sens dans les pages web, et ce afin de pouvoir les retrouver et de donner une représentation de ce sens qui soit exploitable par des agents logiciels. La simple recherche en plein texte ne suffit pas. Les pages web doivent être classées, triées par thème, afin que les mots qu'elles contiennent aient du sens pour des requêtes intelligentes. Quand on cherche à « contacter un avocat sur Paris », on n'est pas vraiment satisfait de tomber sur tous les marchands de primeur de la capitale !

Index

A

accessoire polymorphe 463
adaptabilité 2
adressage 14, 463
adressage des données
  ID/IDREF 142
  les URI 142
  les URL 142
  XInclude 143
  XLink 142
  XPath 143
  XQuery 143
agrégation 63, 463
  de modules de données 100
  définition 96
  exemple 98
  règle de construction 96
agrégation forte 64
ambiguïté 464
analyse
  exemple 293
  top-down et bottom-up 290
analyseur lexico-syntaxique 490
appel d'entité 464
applicabilité 332, 464
  définition 187
arborescence 465
arbre 465
architecture 465
architecture d'entreprise 6
articulation 465
  d'une structure, création 296
association 48
  composition 93
  navigation 99
  références 92
  simple 101
  topique 385
ATA 2100 445
ATA2100 409
attribut
  et type de données 65
  règles de transformation UML/XML 432
  UML transposé à XML 105
  valeur par défaut 435
attributs flottants 465

B

backbone 465
balisage
  applicabilité, sémantique, traitement 188
  orienté document 181
  orienté données 181
base de données 466
base de données XML 15
bien formé 10
BNF 345
BPEL XXI, 12, 446
Brixlogic 11
BURNS 446

C

CALS 183
  exemple (historique) 291
caractère syntaxique 469
cardinalité 466
carte de sujets 466
cartographie de sujets 466
cible 466
classe 467
  abstraite 77
  concrète 77
  contour 74
  dans les DTD 243
  entité 89
  UML 55
classe (RDF) 342
clé 467
clé artificielle 467
clé artificielle, définition 133
clé étrangère 467
collection d'éléments 90
commentaire 460
communauté XML 8
composant 467
composition 467
  association 64
  définition 93
conception d'application 206
conceptuel 468
constructeur 467
correspondance classe UML/élément XML 88
couche de programmation 217
crochet 470
CSS
  exemple 230

D

dans 137
data module code (DMC) 316
data-module code 470
Davenport 241
DC - Development cost 40
DCQ - Development cost with quality 40

déclaration de document 461
définition
adresser une entité 288
attribut migrant 129
contrainte de simultanéité 441
entité 288
granule 288
information élémentaire 288
instancier 389
intégrité référentielle 399
module d'information 288
module documentaire 288
topique 382
workflow 290
démarche de conception 58
dépendance
définition 73
entre schémas 210
représentation graphique 75
dérivation 470, 476
des 125
déterministe 464
diagramme de classe 4
diagramme de classes 470
Digital 241
DMS
définition 220
DocBook 235, 446
document de données 471
document électronique 470
document non structuré 471
Document Object Model 487
document papier 471
document structuré 471
document textuel 471
document XML 2, 471
éléments racines 246
forme canonique 190
forme canonique lexicale 190
forme consolidée 190
forme de base 189
niveau sémantique 172
niveau structurel 172
niveau syntaxique 172
documentation modulaire structurée 472
DOM 446, 487
exemple 224
stockage physique 130
domaine (RDF) 343
donnée 28
unitaire 28
données
d'assemblage 291
DSSSL 446
DTD 472
bi-format SGML/XML 241
classe paramétrée 252
ID IDREF 368
ISO 8879 XXIV
sections marquées 241
stockage physique 132
versus XML Schema 214
Dublin Core 323, 329, 335
DW - Development weight 40

E

effectivité 472
élément
anneau 297
anneau, exemple 298
crochet 297
crochet, exemple 298
global, choisir 88
local, choisir 88
terminal, définition 240
XML 88
élément complémentaire (RDF) 346
élément complexe 257
élément de données 472
SDE 273
élément de données global 472
GDE 273
élément global 473
élément local 473
élément mixte 258, 473
élément purement structurel 473
et analyse top-down 294
règle 296
élément racine 473
élément simple 256
élément vide 260
éléments
articulation d'un modèle 291
éléments flottans 473
ensemble d'information 1
ensemble d'informations 473
ensemble d'informations bien formé 473
entité 474
compagnon dans les DTD 243
paramètre, définition 242
entité document 329
entité non analysée 461
espace de nommage 474
espace de noms 462
espace de noms (RDF) 346
espace de noms, paquetage 84
espaces de noms 474
exemple
CALS 183
composition 94
identification des éléments 212
J2008, schéma dans l'industrie automobile 271
J2EE 207
Java 203
JSP 203
monde des traitements 295
présentation du cas PiloteWeb 59
typage dynamique 80
XLink 214

XQuery 203
XSLT 203
extensibilité 2
extension
XML Schema 83

F

fabrication du contenu 295
factorisation, dans UML 76
feuille 474
d'un arbre 295
définition 240
flexibilité 19
fonction 29
identifier et classer 37
forme 10, 474
PSVI 190
forme canonique 474
forme lexicale 475
forme lexicale canonique 475
forme lexicale normalisée 177
FpML 11, 446
FW - Function weight 40

G

GED 476
GED, architecture 295
généralisation
transformation en relation associative 82
gestion de configuration 475
gestion de contenu 475
gestion de l'applicabilité 475
gestion de versions 475
gestion des connaissances 476
gestion des données 295
gestion des liens 359
gestion des révisions 475
à l'aide d'une base de données relationnelle 427
à l'aide d'une base de données XML 425
à l'aide de XUpdate 426
exemple 409
utilisation d'attributs 402
utilisation de balises fixes 403
utilisation de balises flottantes 404
gestion des révisions et des versions 401
Gleick, James 3
grammaire 476, 485
grammatical (contrôle) 468

H

Hal Computer Systems 241
hardware XML 9
héritage 476
hiérarchie de classe 349
histogramme des fonctions 38
HTML
métadonnées 264
HyTime 381, 447

I

ID/IDREF 14
limitation avec les fichiers 147
limitation dans base XML 166
identifiant 476
identificateur 476
IFX 11
illustration code number 477
imbrication données/traitements 291
information 28
adaptation 32
cycle de vie 29
rythme des modifications 32
Infoset 1, 450
infoset 51, 473, 477
infrastructure 477
infrastructure (des modèles XML) 11
infrastructure UML 2.0 449
instance 51, 478
instancier, définition 76
instruction de traitement 459
intégration 2
ISO 12
ISO 3166 338
ISO 639-1 338
ISO 639-2 338
ISO 8879
1986 480
ISO 8879
DTD XXIV

J

J2008 447
Java
exemple 222
Jelliffe 481
JSP XXI
exemple 225

K

Kant XXV
key 14
keyref 14
Knowledge Management 476

L

langage de liaison 479
langage de modélisation 481
langage de programmation 12
langage formel
définition 47
lexical (contrôle) 468
lien
biunivoque 77
catégories de lien 360
contrôle avec XML Schema 396
définition 360
élément autonome 101
gestion de configuration 393
gestion des révisions 391, 408

lien logique/lien physique 364
navigabilité 103
simple, ID IDREF 367
utilisation des URN 419
lien entre documents 479
lien intra-document 479
lien logique 362
exemple 363
lien physique 361
limite inférieure de décomposition 308
liste des pages effectives 479
lookahead 464

M

macro-structure, représentation 245
méta
définition 56
métadonnée applicative 332
métadonnée de description 332
métadonnée de gestion 332
métadonnée de rangement 332
métadonnées 263, 323, 418, 479
métalangage 479
métamodèle 56, 480
méthode de préparation du projet 25
comparer les scénarios 40
mise en oeuvre 34
plan de classification des services et fonctions 35
modèle 480
conceptuel 71
de stockage 119
des rôles, définition 290
DocBook, historique 241
extensibilité 236
irrégulier, exemple 293
logique, exemple complet 106
macro-structure, exemple 238
modulaire 235
modulaire, représentation 236
physique vs logique 119
physique, conception 120
modèle (concept de) 2
modèle conceptuel 4, 47, 58
de PiloteWeb 60
modèle d'affichage 214
modèle d'association 48
modèle d'objets UML 51
modèle hiérarchique 48
modèle logique 5, 58, 481
modèle père 480
modèle physique 5, 58
modèle RDF 343
modèle relationnel 8
modularisation
des schémas XML 246
modularité
composition d'éléments 248
DTD 239
DTD, éléments racines 239
feuilles de styles 239
schémas XML, déclarations globales 246
module 482
applicabilité 303
articulations et liaisons 70
balisage 306
classification, exemple 317
conception 300
d'information, conception 287
de données 288
de données, définition 70
définition de la S1000D 316
définition de la taille 289
emboîtement 301
équilibre entre contraintes fonctionnelles et techniques 289
identification des éléments 308
lien point à point 309
limite inférieure 308
processus 289
références croisées 308
règles de conception 288
module d'information 255
MOF 57
multiplicité 432
mutualisation
des composants, définition 289
des composants, limite 308

N

navigation
association 91
nœud 482
nombre d'occurrences 483
notation 461
notation BNF 346
Novell 241

O

O'Reilly & Associates 241
OASIS 13
objet
importé par un paquetage 85
objet de modélisation
définition 85
objet métier 17, 483
occurrence 483
occurrence de classe, définition 76
OFX 11
OMG 57
ontologie 483
optimisation du schéma XML 89
Oracle XML DB
répertoires 152
résolution des liens 125

organisation humaine 6
OSF 241

P

paquetage
 définition 73
 individuel 84
 objets de modélisation détenus par 85
 types d'objets importés 85
parcours d'un arbre 486
parseur 325, 483, 490
particule unique 464
pérennité 19
persistance 483
polymorphe 484
polymorphisme 484
Post Schema Validation Infoset 324, 478, 490
processus 295
producteurs des données 295
programmation en couche 208
programme
 d'assemblage 295
 de diffusion 295
 de publication 295
propriété (RDF) 342
PSE (élément purement structurel) 296
PSE, élément purement structurel 270
PSVI 324, 478, 490
 définition 177
publication module code 484
PURL 447

R

racine
 d'un arbre 295
RDF 14, 323, 329, 341, 447
 définition 52
rédacteurs techniques 295
référence 484
 à un objet 91
référence d'entité 484
référence d'entité non résolue 460
règle
 de transformation d'attribut UML en XML 432
 ordre des éléments en UML et XML Schema 443
 séparation données/traitement 188
réification 484
relation
 association 61
 associative 71
 de dépendance 73
 de détention 86
 de généralisation, définition 75
 sous-type 61
 sur/sous-type 77
relation de type spécialisation-généralisation 349
relation élémentaire (RDF) 344
relation multivaluée 349
relationnel 127
Relax 447
Relax NG XXI, 12, 448
RelaxNG 57
requête SQL
 exemple 154, 163
Resource Description Framework 341
ressource 342, 418, 485
 définition 361
 serveur de ressources 418
 topique 383
restriction
 XML Schema 83
RFC 1766 338
RFC 3066 338
rôle d'association 48

S

S1000D 235, 329
 exemple de structure d'assemblage 315
 mécanismes de composition 248
S1000D v2.1 445
SAX, stockage DOM 132
schéma 485
schéma à base de règles 485
schéma conceptuel 480, 485
schéma relationnel
 exemple complet 116
Schematron XXI, 12, 57, 448, 485
sémantique 171, 485
 mode d'expression de 177
 nommage des éléments 173, 186
 type et sémantique 173
 utilisation des attributs 185
sémantique d'un lien 91
sérialisation 486
sérialisé 488
serveur de ressources 488
service 29
 identifier et classer 39
SGML XXIV, 241, 448
SOAP XXI, 353, 448
source 488
sous-schéma, définition 198
sous-type, UML 75
squelette de publication 488
stockage
 adressage des données 142
 dans fichiers 125
 dans fichiers, exemple 147
 document bien formé 127
 DOM, points faibles 132
 DOM, points forts 131
 DTD, points faibles 134
 DTD, points forts 134
 en base XML, exemple 165
 espaces de noms 126

exemple complet 143
méthode DOM 130
mise à jour des données 164
modèles mixtes 126
performance 129
physique, formes de stockage 121
recherche de fragments 129
règles de balisage 182
relationnel, exemple 150
vues relationnelles 154, 157
XML dans base de données XML 139
XML dans relationnel 125
structure arborescente 5
structure d'assemblage 488
analyse top-down 311
diagramme UML 315
exemple 315, 318
superstructure UML 2.0 449
superstructure UML 2.0 488
sur-type, UML 75
SW - Software weight 40
SwiftML 11
syntaxe abstraite 488
syntaxe concrète 489
syntaxique (contrôle) 468
système d'information
flexibilité 31
représentation 33
système d'information 5, 489
représentation 29
système de traitement de l'information 28

T

tableau d'éléments de type array 353
tableau d'éléments de type struct 355
tableau d'éléments multidimensionnel 356
taxinomie 489
taxonomie 489
théorie des éléments purement structurels 253
élément de données 266
élément de données global 266
élément purement structurel 266
règles d'analyse 267
rôle des attributs 261
thésaurus 483
Topic Map 14
Topics Map 380, 449, 466, 489
topique 466, 489
occurrence 389
réification 382
traitement
nommage des balises 173
séparer données et traitement 172, 182
traitements 295
transformation
choix du langage de programmation 209
des schémas UML en XML 89
règle de l'OMG 89
transposition des attributs UML, règles 105
TREX 449
typage
applicabilité 188
dynamique 78
dynamique, attribut xs:type 79
dynamique, mise en œuvre 79
sémantique 188
statique 78
traitement 188
typage dynamique 261
type d'un élément 489
type de données 173
définition 65
type global, choisir 88
type(RDF) 343

U

UBL 13, 449
UDDI XXI, 449
UML 4
attribut, valeur par défaut 435
attributs et types listes 432
classe paramétrée 252
commentaire d'un objet 435
contrainte d'exclusion 439
contrainte de simultanéité 440
élément de modélisation 85
objet importé 85
règles avancées pour XML schema 431
relation de traçabilité 73
stéréotype 442
UML 2.0 477
unique 14
unité d'information 52, 457, 477, 490
unité documentaire 490
unité logique d'information 28
unité d'information, définition 288
urbanisation 490
URI
adresser une entité 288
dans modèle physique 120
définition 362
et stockage dans fichiers XML 125
URL
absolue 361
dans modèle physique 120
définition 361
et stockage dans fichiers XML 125
liens point à point 309
stockage dans fichiers 147
URN 14, 417, 450

exemple 418
URN dans modèle physique 120
UUID 450

V

validateur 490
validation 464, 491
validation de schéma 490
définition 177
validation sémantique 491
vocabulaire 4, 9, 491

W

W3C 8, 12
Web sémantique 341, 491
WebDAV 450
WebDAV, dans Oracle XML DB 155
workflow
vs module, définition 289
WSDL XXI, 12, 450

X

xHTML 333
XInclude
dans modèle physique 120
exemple 300
forme consolidée d'un document XML 190
XLink 166, 368
arc 375
dans modèle physique 120
et modèle relationnel 126
exemple 97, 377
lien étendu 373
lien simle 371
locator 374
modèle conceptuel 370
resource 374
stockage en base XML 169
title 376
variante d'un schéma XML 200
Xlink 11
XMI 450
XML 127, 450
dualité avec UML XXII
langage universel XIX, XXI
outil de modélisation XXI
produire des schémas XXI
type 88
XML Canonical 450
XML Namespace 451
XML Schema 11, 57, 451
dérivation 88, 195
éléments racines 246
extension, restriction, dérivation 83
groupe d'attributs 435
groupe et classe paramétrée 252
mécanismes d'inclusion et redéfinition 193
mixité des modèles 438
pas d'entités paramètres 244
stockage physique 134
type complexe 135, 173
type hérité 136
type simple 135, 173
variante d'un sous-schéma 198
variante par inclusion 195
variante par redéfinition 194
variante, conséquence sur les liens 199
variante, exemple 191
variante, identification 191
variantes 189
XPath 451
dans modèle physique 120
exemple 164
liens dans relationnel 125
mutualisation des composants 308
XPointer
dans modèle physique 120
et modèle relationnel 126
liens dans relationnel 125
mutualisation des composants 308
XQuery XXI, 12, 451
dans modèle physique 120
et stockage dans fichiers XML 125
exemple 142, 217
stockage et DOM 169
XSL 451
XSL-fo 451
XSLT XXI, 12, 451
exemple 227
XSLT 1.0 451
XTM 14
définition 380
XUpdate 12, 452

Dépôt légal : janvier 2006
N° d'éditeur : 7354